(continued from part 44)

Television and film

Many branches of the visual communications media now make extensive use of computers to generate, manipulate and enhance photographic and video images.

Computerised graphics systems have found themselves at home amongst the technology of film and video production. Electronic caption generators have been in use in television for some years, and have now reached high levels of sophistication. These devices can create captions and credit sequences in almost any typeface, colour or size – simply by keying in the correct commands. An extension of the caption generator has also been developed to provide television graphic artists with a tool for generating more exciting logos: for instance, the new BBC 2 logo was produced on such a machine. (Channel 4's logo was produced on a more complex machine).

Developments of this type have led to the digital TV picture manipulation equipment now in common use. For example, the **Quantel** system can reduce, enlarge, dissolve, overlap, merge, flip, repeat and distort television pictures in seemingly unlimited ways. This is achieved by digitising the video signal and then manipulating it under computer control.

In today's science fiction films, computer images seem to be overwhelming most of the action. However, these 'computer graphics' are usually carefully animated drawings, optically enhanced to look like computer generated graphics. The computer game inspired movie

Below: this picture was produced using the Quantel Graphic Paint Box. The image was created on screen, output as hard copy and then printed on a five colour press.
(Photo: Micro Consultants/Quantel).

'TRON', though, did use computer generated images. These were matted together with live action and specially shot sets to create a microchip world. This type of technique is being used more and more for animation in both films and advertising.

The computer provides the capability to melt moving images in a way that previously needed the skilful creation of hundreds of thousands of individual frames – each one drawn and coloured by hand. Now the animator uses computers to create, draw and store the dynamics of movement, background and colouring.

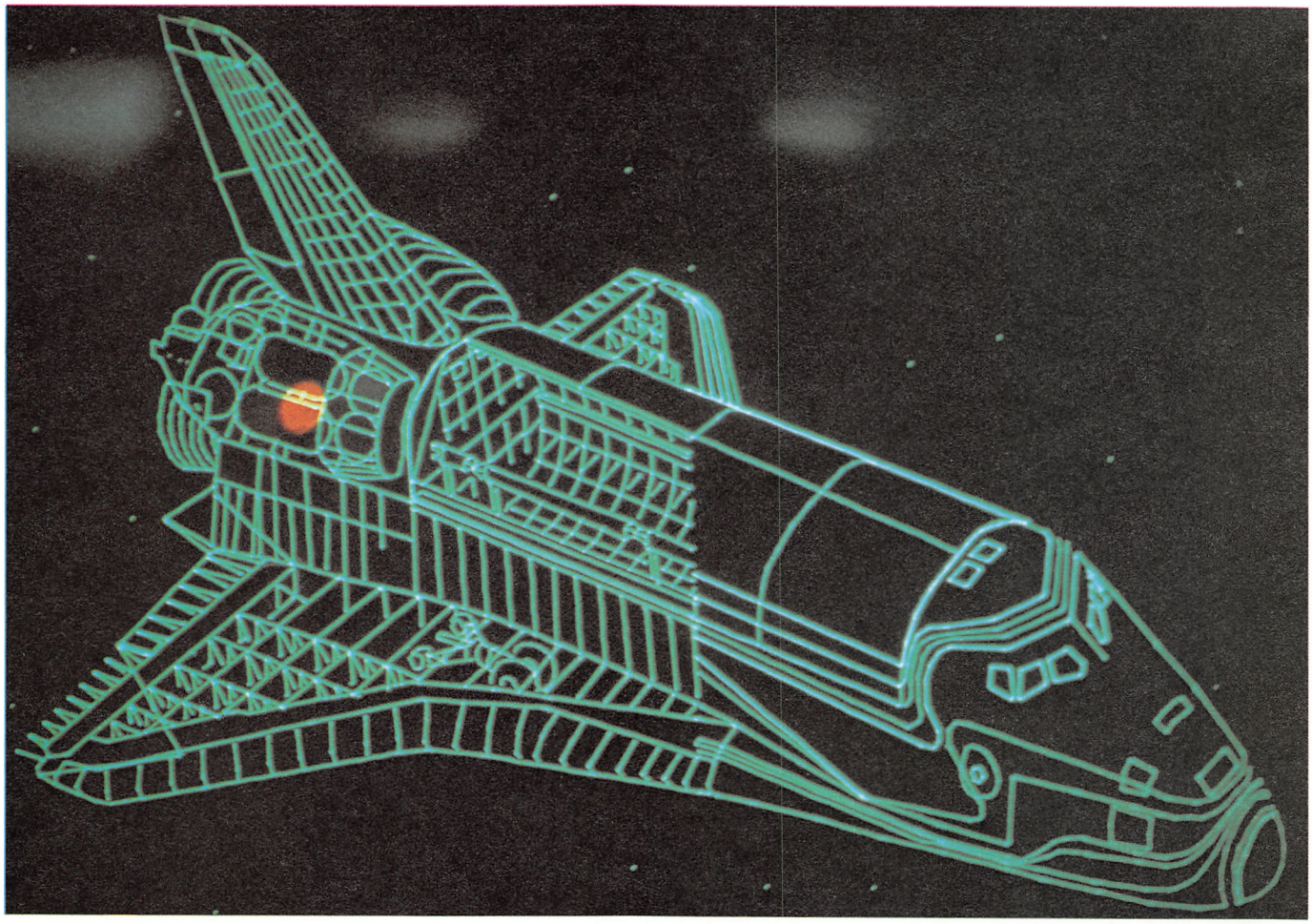
Simply by using a light pen, a character can be drawn on a display screen, copied and changed into another position. In this way the different stages, the **key frames**, involved in animating, say, a man walking, can be drawn. The computer then automatically fills in the frames inbetween, enabling the simulation of smooth movement. Starting from a basic 'stick man' drawing, then, the animator is thus able to develop the movement of the characters and objects – then add the actual forms of the objects: texture, faces, expressions, colour and the effects of light and shade (chiaroscuro). This process also allows animators the freedom to develop and alter their work, right up to the final stage of recording onto video tape or film.

Is it real?

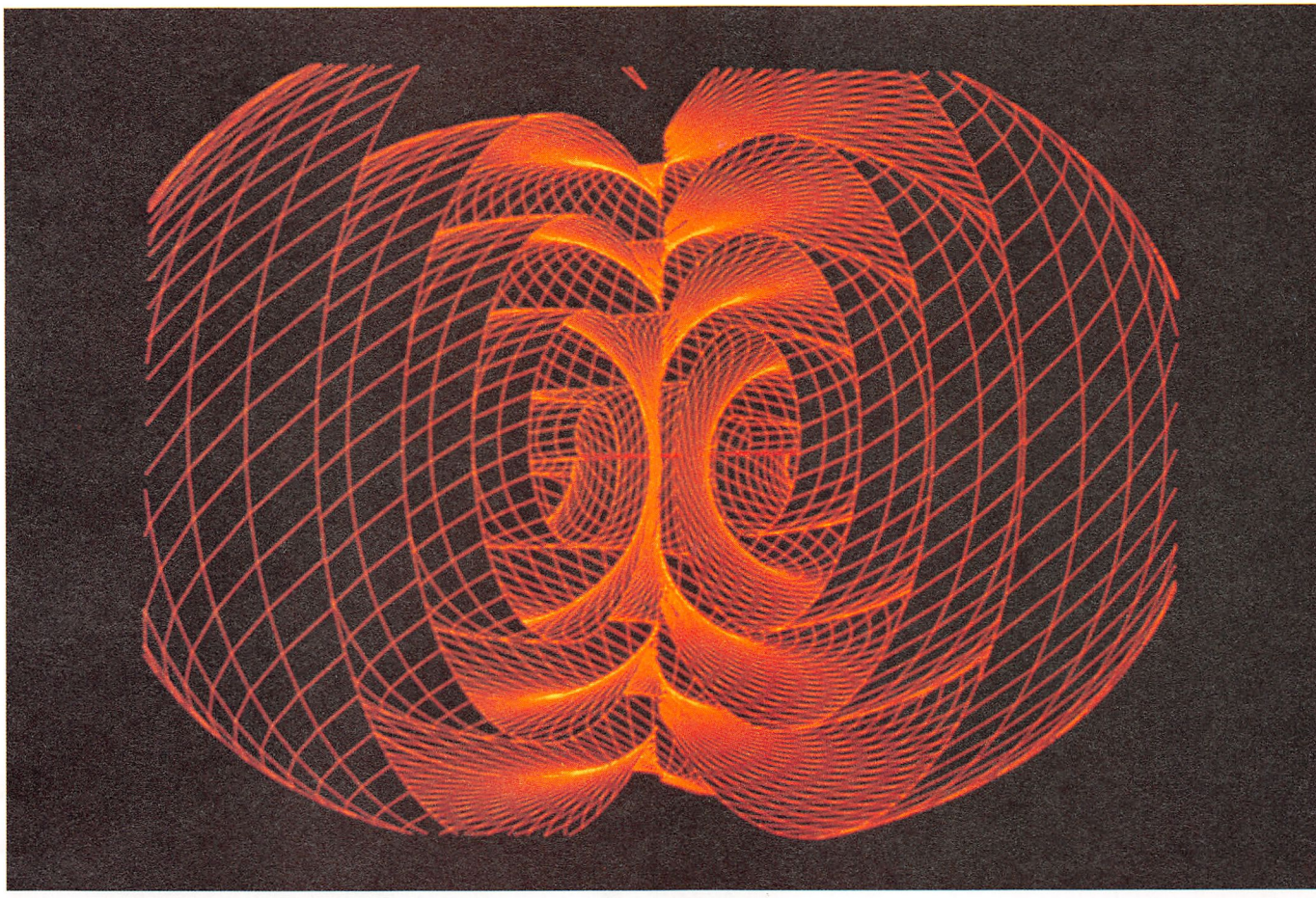
One very important application for computers graphics lies in simulation. Both civilian and military pilots are now trained on flight simulators. These provide life-sized, realistic models of aircraft cockpits that can be 'flown' through a computer generated visual world. The high resolution graphics used in such systems, allied to operational controls that manoeuvre the 'plane' through the sky, and motors that twist and tilt the cockpit, provide a realism that can induce airsickness.

Very powerful computers are necessary to run simulators like this. They have

Below: computer generated 'wire diagram' of the space shuttle.



Science Photo Library/Jerry Mason



Science Photo Library/Dr. A Winfree

Above: this image was created on a CRAY–1 supercomputer and represents the wavefront of any excitable medium.

to be able to provide very high resolution, multicoloured graphics, run the control interfaces and to support situation modelling.

The display presents a real-time animated picture – as the plane flies through the 'sky', the view must be seen to react according to the pilot's actions and instructions via the flight panel. The image, therefore, has to be updated many tens of times a second, to ensure continuous animation.

Medical and scientific applications

Thermographic imaging is based on the principle that the temperature of the skin varies depending on the specific cellular or circulatory processes occuring inside the body at any point. In thermography, skin temperatures are measured and then represented visually by different colours. This is a particularly useful medical tool which can be used to detect tumours and other disorders by analysing the presence or absence of heat in the body.

Thermography is also useful in sports medicine – enabling doctors to study the activity of various muscles and their oxygen requirements at different stages of a sprint for example.

The study of the characteristics of movement in this way can also aid physiotherapists to develop individual programmes of treatment for spinal muscular disorders.

Graphics images are also being used by mathematicians at the University of Bremen, in West Germany, to help solve complex, baffling equations. They have managed to link the mathematical theory of what are known as 'Julia sets' (proposed by the French mathematician, Gaston Julia, more than sixty years ago) with the physical theory of phase transitions (for example, the boiling of water or melting of ice).

Although much more work needs to be done, computer images are providing new impetus in a field of research that has baffled mathematicians for years!

Market changes

The computer graphics market has changed from a low key, rather esoteric venture into a thriving commercial concern. Certainly the increased sophistication of microcomputer packages has helped, but this is by no means the whole story. The use of dedicated graphics systems is on the increase too, and is currently growing by some 35% per year in the U.K. CAD/CAM equipment sales alone now generate about £150 million per year, even though they are being used by less than 1% of British industry.

As graphics systems will inevitably become cheaper – one company, Valid, which markets systems for semiconductor design, has already announced a new version of an existing system, for around one fifth of the present price – their usage will increase.

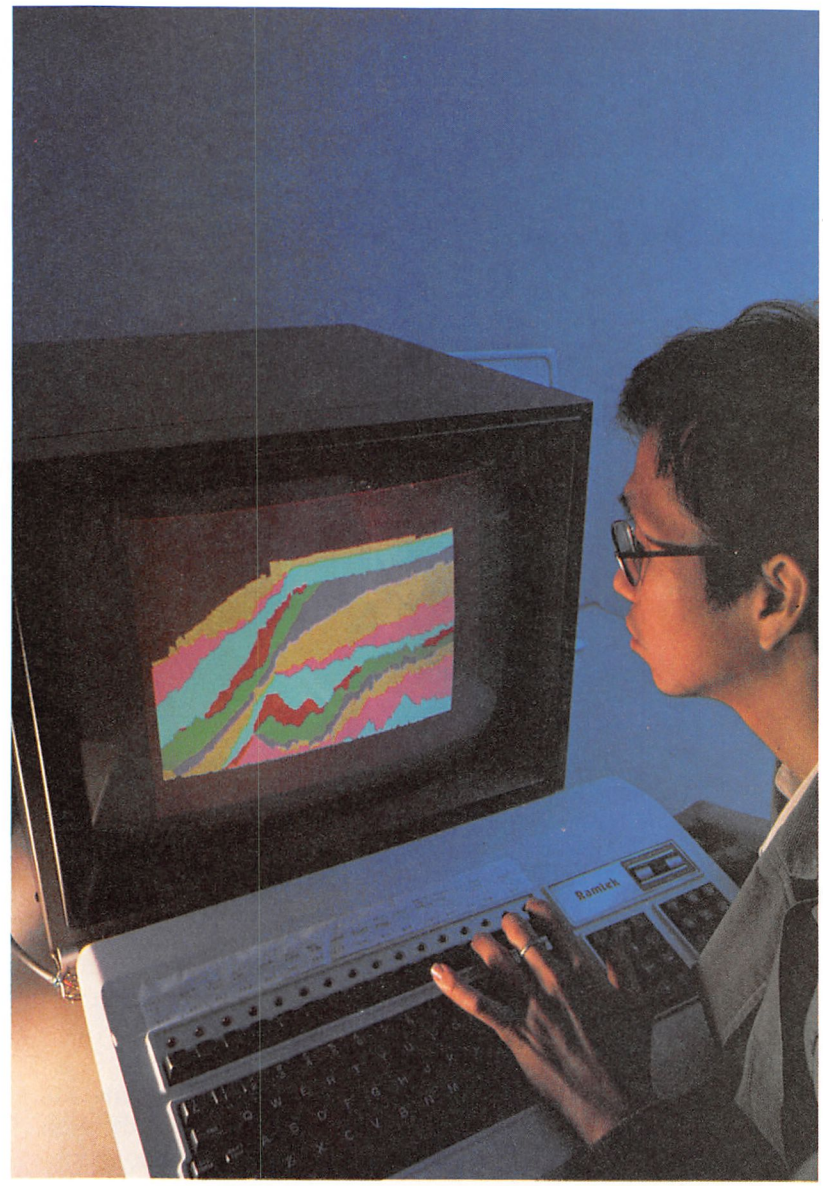
This reduction in price is being considerably aided by improvements in the power of microprocessors. National Semiconductor, Motorola and Zilog are all engaged in using new CMOS designs giving their products a two or three fold increase in power. It is expected that a microprocessor with a 10 mips (millions of instructions per second) performance will be available by the middle of the next decade.

For comparison, this is equivalent to a fairly large mainframe processor in use today. Moreover, the cost of this new microprocessor will be only a fraction of the price of the mainframe processor – perhaps as low as one thousandth!

Systems bought in the future are unlikely to be dedicated graphics systems (though there will still be a market for these in very specialised areas), leaving the main emphasis on the development of software and databases. It is likely that centres of expertise will spring up, each specialising in a particular area of software design and development.

In fact, this is already happening. Established with funds from the British Technology Group in 1977 (and changing both its name and nature several times since then), the Cambridge CAD centre is now a thriving place for engineering research into graphics based systems. The centre now engages in the research and development of a range of specialised software tools, including the GEMS image processing packages and the PDMS plant design programs. Both of these packages will run on a variety of machines.

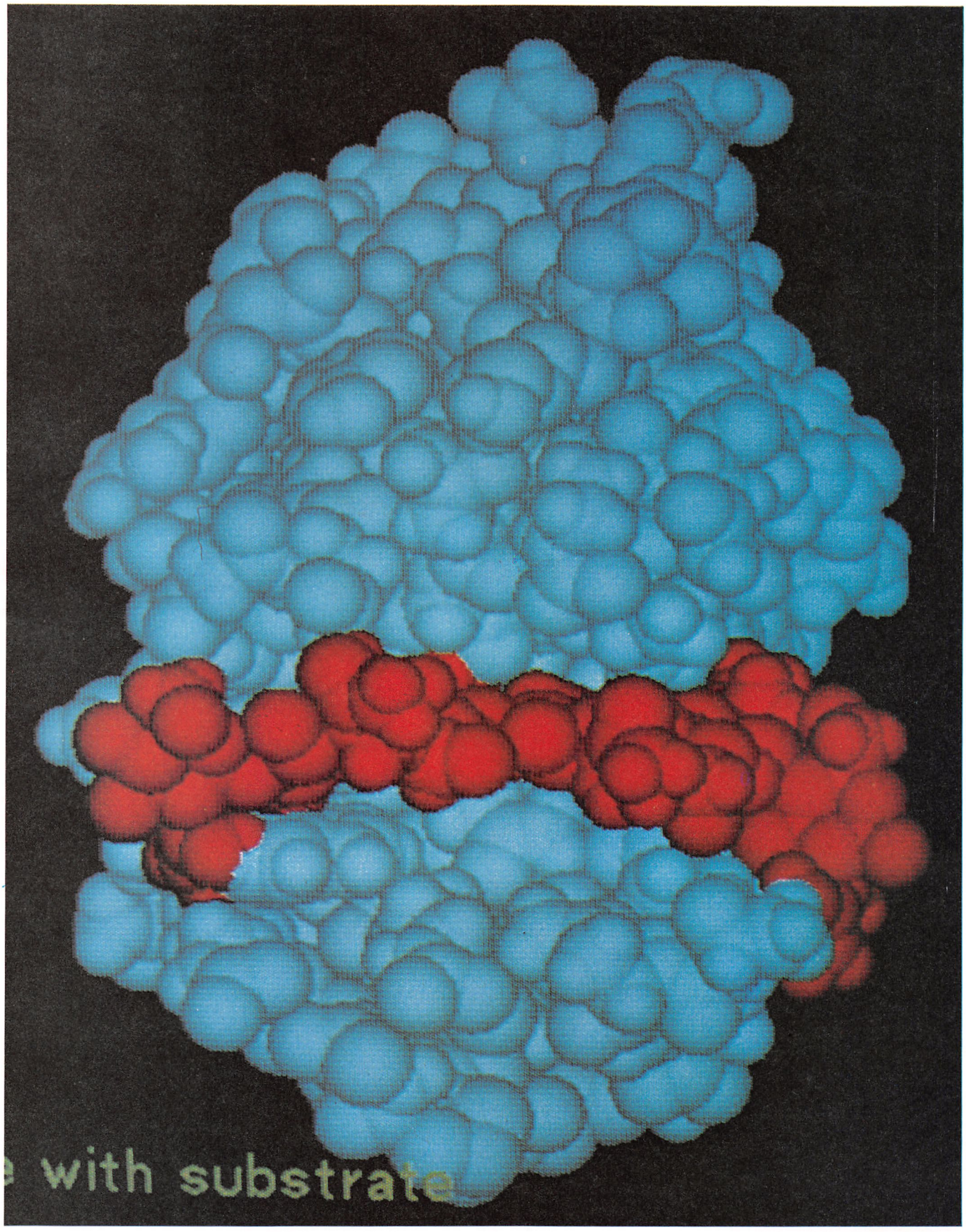
There is also the coming prospect of portable machines, with screens capable of supporting high standards of resolution. Japanese manufacturers, such as Sharp and Hitachi, are soon to release liquid crystal flat-screen displays which rival the conventional cathode ray tube. These screens are no bigger than an A4 sheet of paper and weigh just eight ounces. Bigger flat screens are on the way offering even greater resolution.



Science Photo Library/Hank Morgan

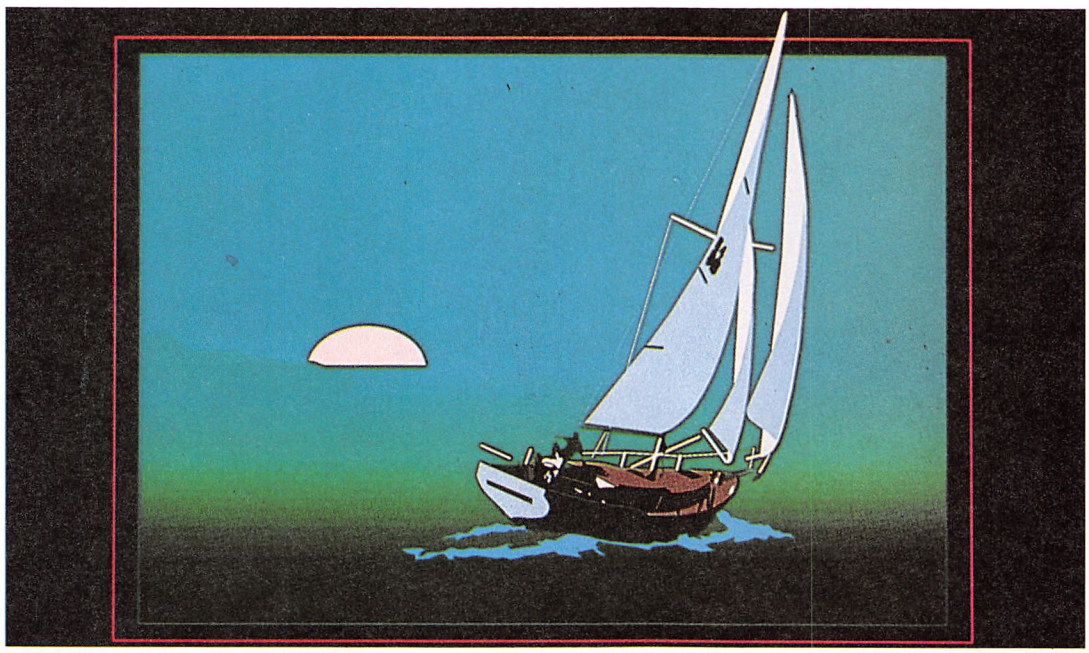
Above: here, at the Exxon Research Laboratories, a supercomputer is used for seismic sequence analysis during the search for land formations which might bear oil.

Right: computer generated image of egg white lysozyme attached to its substrate.



Science Photo Library/Division of Computer Research and Technology, National Institute of Health

Left: this computer generated image of a cityscape was first presented at 'SIGGRAPH '81' – the world conference on computer graphics.



Left: another computer image, this time from 'SIGGRAPH '83'. (Photo: Association for Computing Machinery).

Conclusion

There are opposing views as to the impact of the increased use and sophistication of imaging systems on everyday life. On the one hand is the view that computer aided design and simulation techniques will lead to a greater variety of products from which to choose. These products all having a shorter span of useful life, but being more reliable and efficient during their *intended* life. The view stems from the fact that designing and testing of those designs will become a faster, cheaper process.

Opposing this view is the feeling that fewer, better designed products which are more efficient and last longer should be the end result. Probably, though the greater financial gains to be made from the former approach mean that this view is likely to predominate.

One area which undoubtedly will be affected will be the leisure industry. In the future, travel may no longer by necessary – we'll take our holidays by sitting in an 'event simulator'. This could mean experiencing ancient Rome or the Wild West, participating in the Olympics or just 'seeing the U.S. from coast to coast'.

Where physical travel was necessary, hotels could be chosen by 'wandering around them' on the T.V. screen – comparing facilities and location.

Concepts of programming

Developing programs

As we have seen from *Microprocessors 7* and *8*, programming is a basic requirement in all microprocessor applications. Developing a program is just like developing any other type of design, in terms of the sequence of tasks that have to be performed.

The basic operations include:

1) Write a general description of the desired overall system performance.
2) Identify the overall system inputs, outputs, and general subsystem operations.
3) Describe each subsystem operation, identifying inputs, outputs, and the tasks involved.
4) Continue subdividing system tasks and developing task descriptions until they are defined at the most elementary level.
5) Write the instruction sequences that implement all the elementary subsystem tasks.
6) Combine the individual subsystem task instruction sequences into the desired overall system program.

Program operations are best described as lists or flowcharts such as that shown in *figure 1*. You'll notice that the mnemonic forms used in the list are mostly abbreviated sentences, but they may be **macro-mnemonics** that correspond to actual high level program instructions.

Subprogram modules

The second stage in program development is the division of the entire system into a number of subsystems or program modules. This provides an organised approach to analysing each subsystem task, so that its requirements can be quickly recognised and described. Each subsystem can be thought of in terms of the model shown in *figure 2*: a process or operation is to be performed on incoming data and the results given as output.

Modular programming

Once the different tasks of the system have been identified, then the next step is to describe them in detail. This modular programming procedure is sometimes not necessary when developing simple systems, but is indispensable for the proper development of larger programs.

The description of each task then has to be turned into a procedure or algorithm that will meet the task's requirements, and thus provide a solution to the problem. The **algorithm** must be in sufficient detail to enable the final instruction sequences to be written.

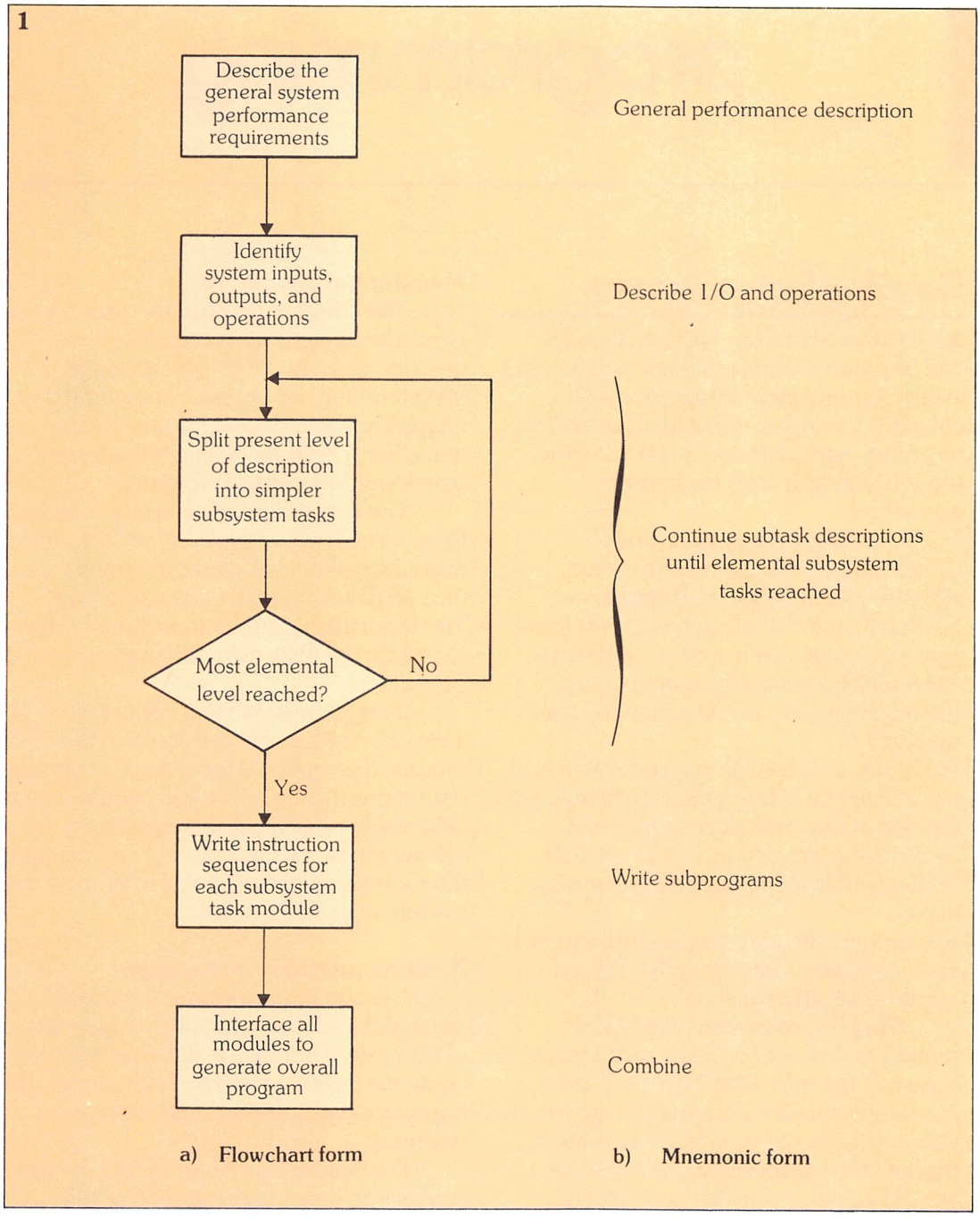
It might be possible to write the instruction sequence directly from the module description. However, for complex operations the algorithm will need to be described in detail before it can be implemented as a subprogram. In either case, the modular approach helps to simplify the overall programming task.

Relating programs to hardware

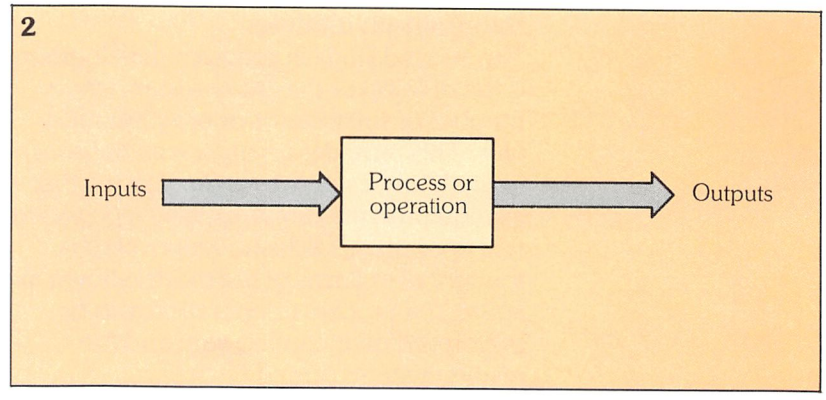
So far, program development has been discussed as if it were totally independent of the system hardware. If the system is to complete its task in an efficient manner, the hardware and software must work in harmony.

For example, in order for the microprocessor to direct the operation of the system components correctly, it must properly identify the parts at the appropriate time and in the right sequence. It does this with addresses and control and timing signals. As a result, one of the first tasks for the hardware designer is to choose certain address ranges for both the program memory and the data memory.

In many cases, these are chosen to simplify the hardware circuits required to interpret or decode the addresses coming from the microprocessor. In other cases, these addresses are defined by the microp-



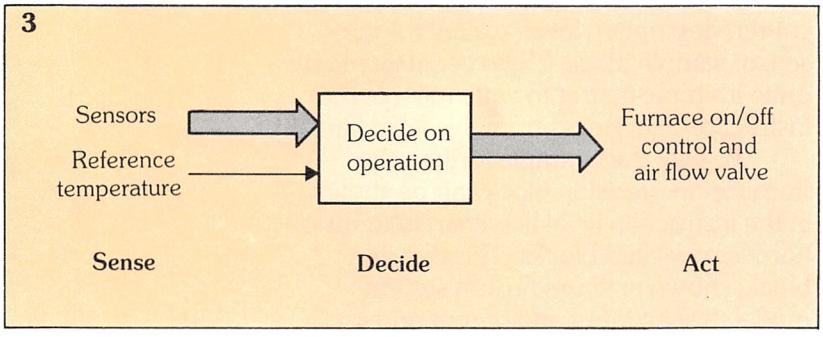
1. (a) Flowchart describing the basic operations involved in writing a microprocessor program; **(b)** the procedure described in mnemonic form.



2. General subsystem model for program descriptions.

rocessor and the designer has no choice in the matter. For example, the location of interrupt sequence subprograms in many microprocessor based systems are at fixed addresses. The exact address varies depending on the microprocessor or microcomputer used. Similarly, certain microprocessors specify some input/output subsystems to certain ranges of addresses.

Throughout system operation, the program must therefore specify the correct addresses for relevant instructions. In this

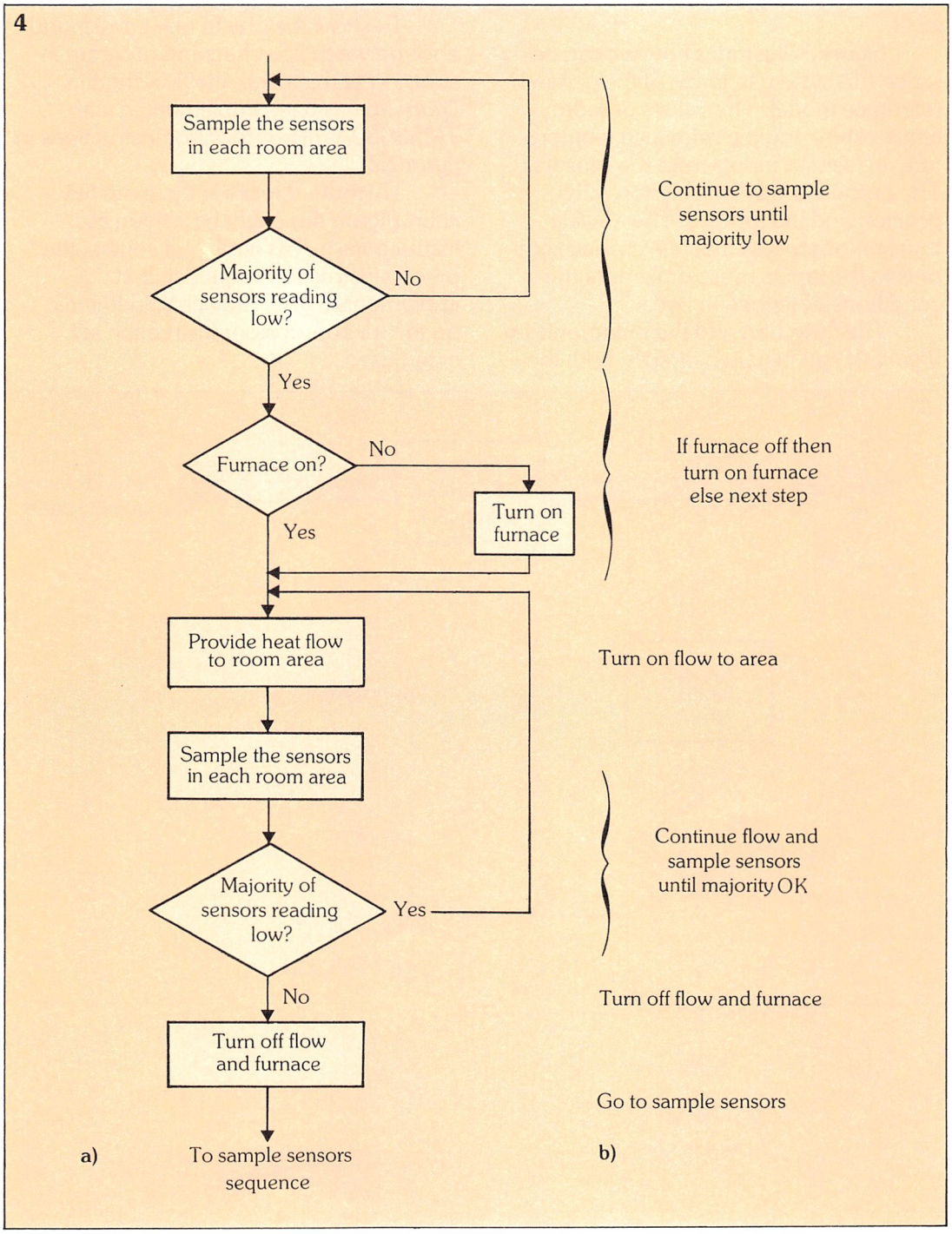


way the data to go with these instructions, the interrupt sequence subprograms, and the I/O subsystems do not overlap.

When all of these factors have been considered, addresses are assigned to the various memory and remaining I/O subsystems in a process known as **memory mapping** – since it maps or relates the devices to the addresses that the program must use to locate and interact with them.

3. A computer controlled heating system described in terms of the reference model of *figure 2*. Note that this conforms to the Universal Systems Organisation that we have met many times before.

4. (a) Flowchart and (b) mnemonic form description of the computer controlled heating system.



An example

Let's take the example of a computer controlled heating system. There are eight temperature sensors in each area that the computer has to monitor. Once a majority of these sensors fall below a preset reference temperature, the furnace is turned on (if it is not already) and the flow control valve permits hot air to enter that area. This flows until the majority of the sensors indicate a temperature *above* the reference level.

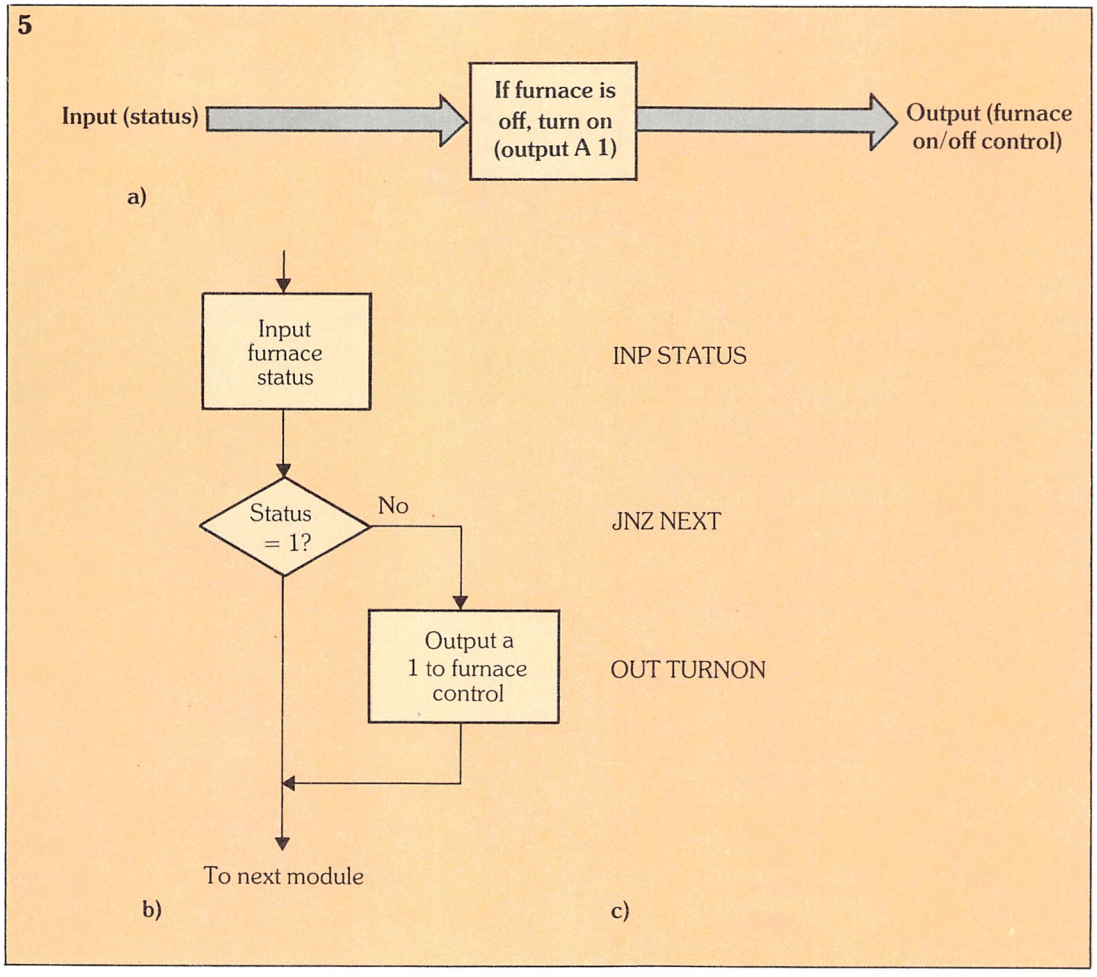
Figure 3 illustrates how we can describe this system in terms of the general reference model. The values of room temperature and the reference temperature are sent as inputs from the sensors. The outputs control the air flow valve position and the furnace. The 'decide' function or process that relates these outputs to the inputs must implement the conditions described above.

The flowchart and the mnemonic list (*figure 4*) can be taken directly from the written description level, except for the sensor sample block. However, it would be quite a simple matter to write most of the instruction sequences from this flowchart.

To take the example further, the 'furnace on' decision block can be shown in the instruction level flowchart in terms of hardware related blocks. This decision block, shown in *figure 5a* as a system model, has an input and output and a simple decision making process.

To show the one to one correspondence between flowchart and program instruction at this stage, the flowchart in *figure 5b* has been implemented with TMS8080A microprocessor instructions in *figure 5c*.

The other blocks in the main flowchart (*figure 4a*) could be written as instructions just as easily. Of course, the precise instruction sequence that implements a given flowchart depends on the microprocessor instruction set being used.



5. Development of the 'furnace on' decision block of *figure 4:* **(a)** system model; **(b)** flowchart description; **(c)** TMS8080A microprocessor instructions.

How do instruction sets differ?

Each family of microprocessors has its own unique instruction set. However, there are, of course, some similarities between them. All microprocessors support the instruction set used by the fictitious SAM (looked at in the previous two *Microprocessor* chapters), but they also handle many other instructions.

sets: 4-bit microprocessors will be represented by the TMS1000, which is basically an extension of SAM – a fact reflected in its instruction set; 8-bit devices will be represented by the TMS8080A instruction set; and 16-bit microprocessors are represented by the TMS9900 family's instructions.

Addressing modes

Table 1 compares the different addressing modes available in the three devices. The

Table 1
Comparison of addressing modes

Addressing Mode	TMS1000	TMS8080A	TMS9900
Immediate	Yes	Yes	Yes
Register	Yes	Yes	Yes
Register indirect	Yes	Yes	Yes
Data counter	Yes	No	Yes
Indexed	No	No	Yes
Direct	No	Yes	Yes

Table 2
Comparison of arithmetic logic instructions

Operation	TMS1000	TMS8080A	TMS9900
Addition:			
incrementing	IMAC,IA,IYC	INR,INX	INC,INCT
without carry	AMAAC,Ac*AAC	ADD,ADI,DAD	A,AB,AI
with carry	—	ADC,ACI	—
Subtraction:			
decrementing	DMAN,DAN,DYN	DCR,DCX	DEC,DECT
without borrow	—	SUB,SUI	S,SB
with borrow	—	SBB,SBI	—
Negation	—	—	NEG
Absolute value	—	—	ABS
Multiplication	—	—	MPY
Division	—	—	DIV
OR	—	ORA,ORI	ORI
AND	—	ANA,ANI	ANDI
Exclusive OR	—	XRA,XRI	XOR
NOT	CPAIZ	CMA	INV
Clear	RBIT,CLA	—	CLR
Set	SBIT	—	SETO
Shifts (S) or	—	RLC,RRC	SLA,SRA
Rotates (R)	—	RAL,RAR	SRC,SRL

**Displacement 6, 8 or 10*

Generally speaking, the more bits that a microprocessor can handle at one time, the more instructions it can support. To illustrate some of the typical differences between microprocessors, we'll examine summaries of three different instruction

16-bit unit supports all addressing modes currently used by microprocessors; the 8-bit unit supports all but indexed addressing; and the TMS1000 4-bit unit offers only limited addressing options. As a result, it is much more difficult to write a program

Table 3
Comparison of data movement instructions

Operation	TMS1000	TMS8080A	TMS9900
Input	TKA	INP	TBIT,STCR
Output	SETR,RSTR, TDO,CLO	OUT	SBO,SBZ,LDCR
Memory-register and register-register	TAY,TYA TAM,TMA TMY,TYM TAMZA,TAMIY, XMA	MDV,LDA,STA LDAX,STAX LHLD,SHLD SPHL	MOV,MOVB
Constants	TCY,TCMIY LDP,LDX	MVI,LXI	LI,LWPI,LIMI
Stack operations	—	PUSH,POP, XTHL	—

using the TMS1000 unit than the TMS9900 unit, not just because it is a 4-bit microprocessor, but simply because the TMS1000 often makes it more awkward to locate needed data.

This is a problem that has already been experienced with SAM, but it is important to realise that the TMS1000 is a self-contained microcomputer, with built-in memory and it is therefore limited to using what memory is available.

Arithmetic logic instructions

The three microprocessors' arithmetic logic instructions are compared in *table 2*. While all three microprocessors have increment, add, decrement and subtract instructions, only the TMS8080A supports add with carry and subtract with borrow.

The 16-bit device can handle all the arithmetic and logic operations listed, including absolute value, multiplication and division. Multiplication and division can be carried out by this microprocessor using one instruction each, rather than by using a complete sequence of instructions made up for the operation. The TMS1000, on the other hand, has only a limited range of ALU instructions.

Data movement instructions

All three microprocessors under examination here have similar data movement operations (*table 3*). All three can move program constants into registers or memory, and all can provide single or multiple bit input and output instructions. Similarly, they can all move data from registers to and from memory.

The TMS8080A needs **stack control operations** and provides them with its PUSH and POP instructions. All three microprocessors have quite adequate data movement options, and provide about all the options that could be available.

Comparison and branch operations

The **comparison** and **branch instructions** used by the three microprocessors are shown in *table 4*. The TMS1000 is the most limited in this area. It offers less than or equal to comparisons between the accumulator and memory or a constant, and checks to see if the accumulator, Y register, or memory is not zero. The branch (BR) checks a status flip-flop which saves the results of a comparison, or a carry resulting from a previous arithmetic operation. One level of conditional branch is available, depending on the condition of the status flip-flop.

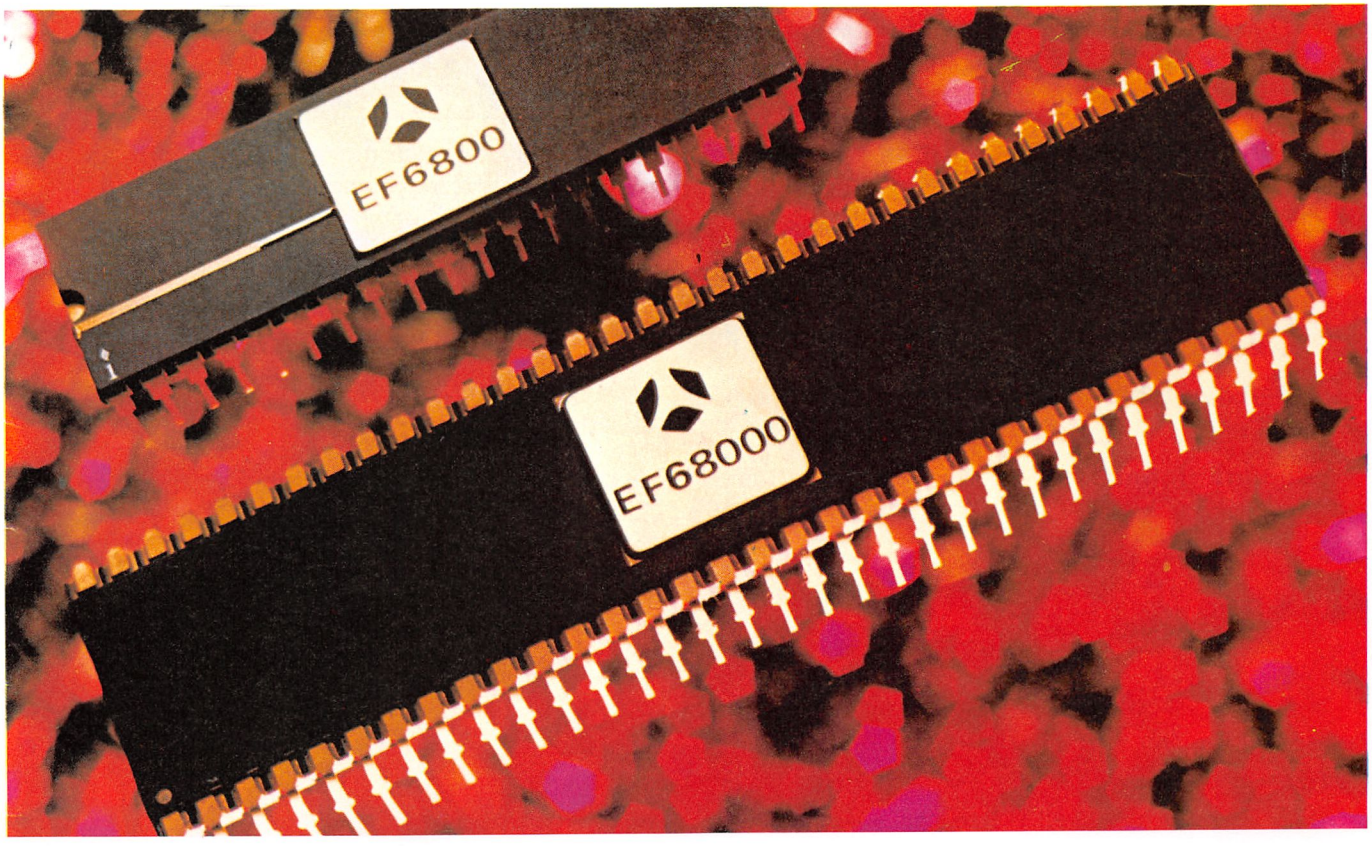
The 8-bit device offers arithmetic comparisons, **unconditional branches**, an unconditional subrouting jump (call) and **conditional branches**, subrouting jumps (calls) and subroutine returns. The condition checks allowed by the TMS8080A include carry, no carry, zero, not zero, plus, minus odd parity and even parity.

The 16-bit TMS9900 offers arithmetic and logical comparisons, the same branch and jump options as the '8080A, as well as additional branch conditions and subroutine calling procedures with a much broader scope. These additional options enable the programmer to implement decision making, subrouting structures, and input/output subprograms more efficiently.

Table 4
Comparison of branch and comparison operations

Operation	TMS1000	TMS8080A	TMS9900
Arithmetic comparison	ALEM,ALEC	CMP,CPI	C,CB,CI
Logical comparison	MNEZ,YNEA YNEC,KNEZ	—	COC,CZC
Unconditional branch	—	JMP,RET	B,JMP
Unconditional subroutine jump	—	CALL	BL,BLWP RTWP,XOP,X
Conditional branch	BR,CALL, RETN Conditioned on status flip-flop	Jcond,Ccond Rcond Conditions: Z–zero NZ–not zero C–carry NC–no carry P–plus M–minus PO–odd parity PE–even parity	Jcond Conditions: EQ = NE = OC Carry NC No carry GT > LT < OP Odd Parity NO No Overflow H Higher Than HE Higher or = LE Lower or = L Lower

Below: two microprocessors in common usage: the 40-pin DIL packaged 8-bit EF6800, and the 64-pin DIL packaged 16-bit EF68000. (Photo: Thomson-CSF).



Summary

The three microprocessors that we have examined here have provided us with an example of the kind of devices currently available, and the wide range of operations that they are presently capable of performing.

A given microprocessor's instruction set can only really be understood by designing systems with it. With this in mind, examples of applications of an 8-bit and 16-bit device will be looked at in the following chapters.

Although the instruction sets for the three types of device rise in capability as the number of bits increases, this does not mean to say that 4-bit microprocessors don't have a place in system design. They do, of course, and the key to efficient design lies in choosing the right microprocessor for the job.

A 16-bit microprocessor processes four times as much data as a 4-bit unit in a given instruction. This means that the 16-bit unit will be at least four times more efficient in processing data and other information than the 4-bit unit.

Similar advantages exist for the 8-bit over the 4-bit microprocessor or the 16-bit over the 8-bit microprocessor. However, many applications do not require the efficiency and power of the 16-bit instruction set – they might only need the capabilities of an 8-bit or a 4-bit unit. Or the case may be that one instruction from the 4-bit microprocessor means that it performs as well as a 16-bit unit in a specific application.

Glossary

algorithm	a procedure or sequence of instructions which enables a microprocessor to perform a complete task
branch instruction	an instruction in a microprocessor's instruction set which can cause program execution to shift to another sequence of instructions
comparison instruction	a microprocessor instruction which enables the microprocessor to compare the conditions of two items of data, then branch if required
conditional branch	a microprocessor instruction enabling the microprocessor to perform a test and go to different addresses for the next instruction, depending on the test results
macro-mnemonic	abbreviations of system processees, corresponding to high level program instructions. Generally, macro-mnemonics are formed by a number of instructions from the microprocessor's instruction set
memory mapping	the process of assigning memory to input/output subsystems and data, which thus relates the devices and data to the program
unconditional branch	an instruction which always causes a transfer of control from one sequence of instructions to another

15

COMMUNICATIONS

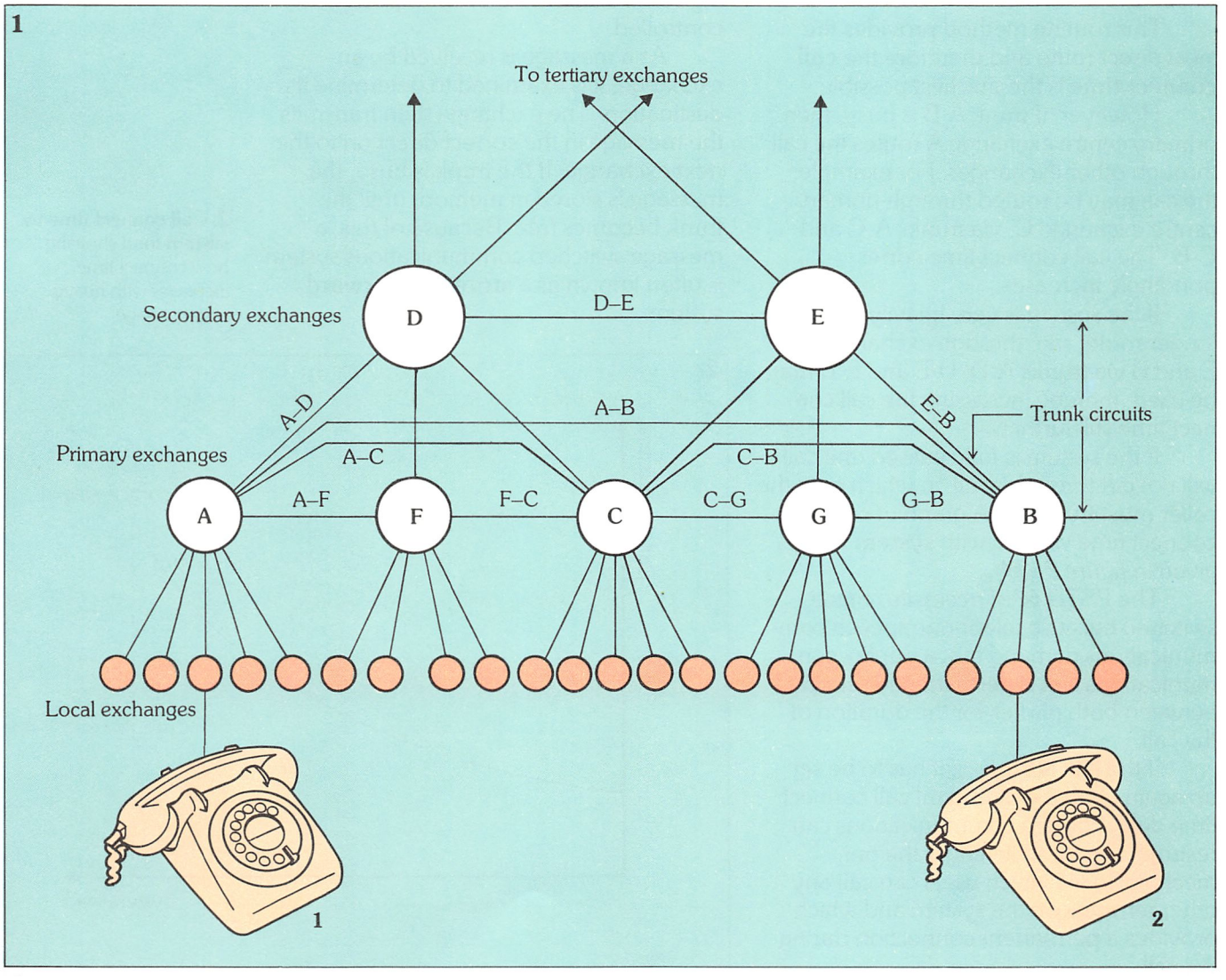
Comparing switching systems

Circuit switching

Communications systems, of course, rely on a connection being made between the information source, or sender, and the receiver: this applies equally to data as well as to speech communications. Thinking back to *Communications 2* and *3*, we know that the PSTN relies on a technique known as **circuit switching** to ensure that this connection is made.

A circuit switched communications system displays the following six features:
1) There are a number of exchanges, each capable of circuit switching;
2) These exchanges are interconnected by a number of point-to-point circuits (trunks);
3) All point-to-point circuits and switches are used for the duration of the call;
4) After a call has been set up, the capacity of the connection is limited (that is, in a

1. The exchange hierarchy and trunk circuitry of a circuit switched communications system.



speech call, the bandwidth is limited; but in a data call, the data signalling rate is limited);
5) Users do not need to know how the circuit switching takes place – they merely need to know how to make the call;
6) As more and more users make calls on the network, so the acceptance of calls is delayed.

The exchange hierarchy and connecting trunk circuitry used in a circuit switched communications system is shown in *figure 1*. If, say, terminal 1 is to be connected to terminal 2, the call may be routed in a number of ways. First, the call is routed through the terminals's local exchange to primary centre exchange A. From here, the call is routed directly to primary centre exchange B via the trunk A-B, and on to terminal 2 through its local exchange.

This routing method provides the most direct route and therefore the **call connect time** is the quickest possible.

However, if trunk A-B is busy, then primary centre exchange A routes the call through other exchanges. For example, the call may be routed through primary centre exchange C, via trunks A-C and C-B. The call connect time correspondingly increases.

If the system is very heavily used, a longer route, say, through exchanges A, D, E and B via trunks A-D, D-E and E-B may be used, thereby increasing the call connect time still further.

If the system is fully utilised, the call will not get through at all, in which case the caller must try again. A graph of call connect time varying with **system load** is given in *figure 2*.

The PSTN is, of necessity, circuit switched because telephone speech communications demand that a duplex communications link is permanently set-up between both parties for the duration of the call.

If the link is broken, it has to be set up again (with the attendant call connect time delay) before communications can resume. Circuit switching is the only mechanism via which users can call any other terminal on the system and which provides a permanent connection during the call.

Message switching

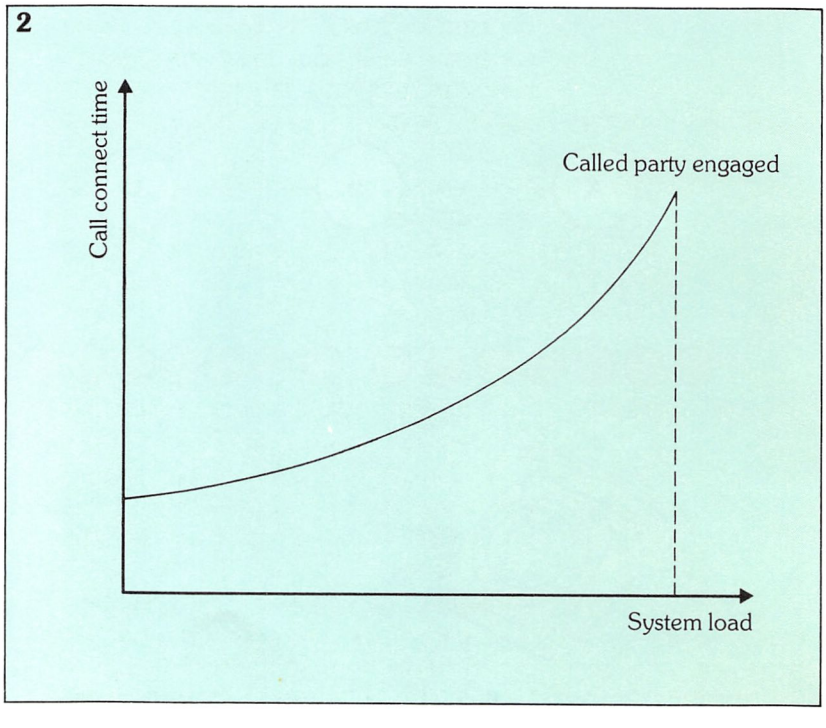
In a data communications system, on the other hand, the provision for permanent connection is no longer necessary. Systems which enable data transmission between users without a permanent connection being set up between the two are known as **message switching** systems.

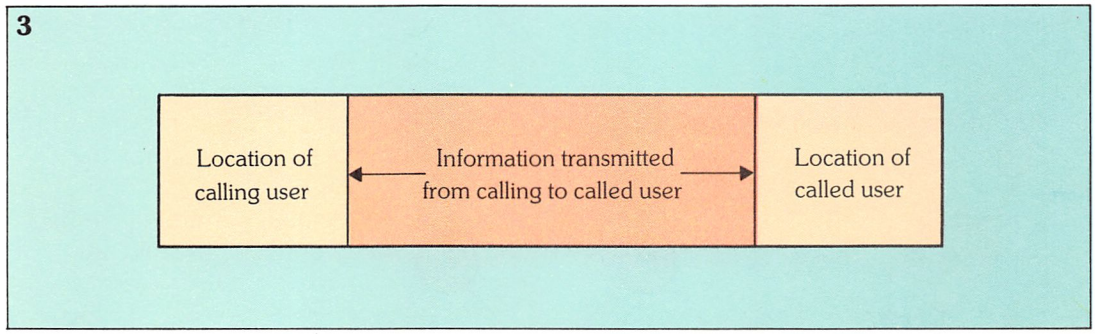
Each message comprises information relating to the calling terminal and the receiving terminal, as well as the data to be communicated; a possible format is shown in *figure 3*.

Message switching is similar to circuit switching in that point-to-point trunk circuits connect a number of switching exchanges (*figure 4*); these exchanges, however, provide a memory facility which is used to store messages. Message switching systems are therefore computer controlled.

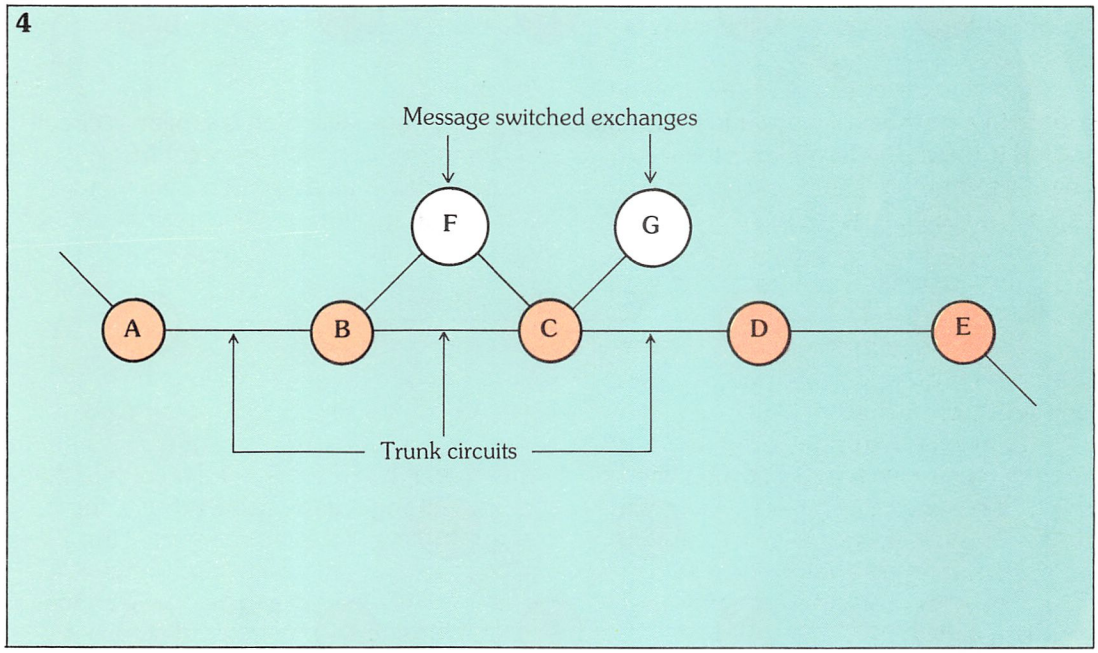
As a message is received by an exchange, it is examined to determine its destination. The exchange then transmits the message in the correct direction to the next exchange. If the trunk is busy, the message is stored in memory until the trunk becomes free. Because of this, a message switched communications system is often known as a **store-and-forward** system.

2. Call connect time *vs* system load showing how connect time increases with heavy system usage.





3. Message switched systems utilise this kind of format for their messages.



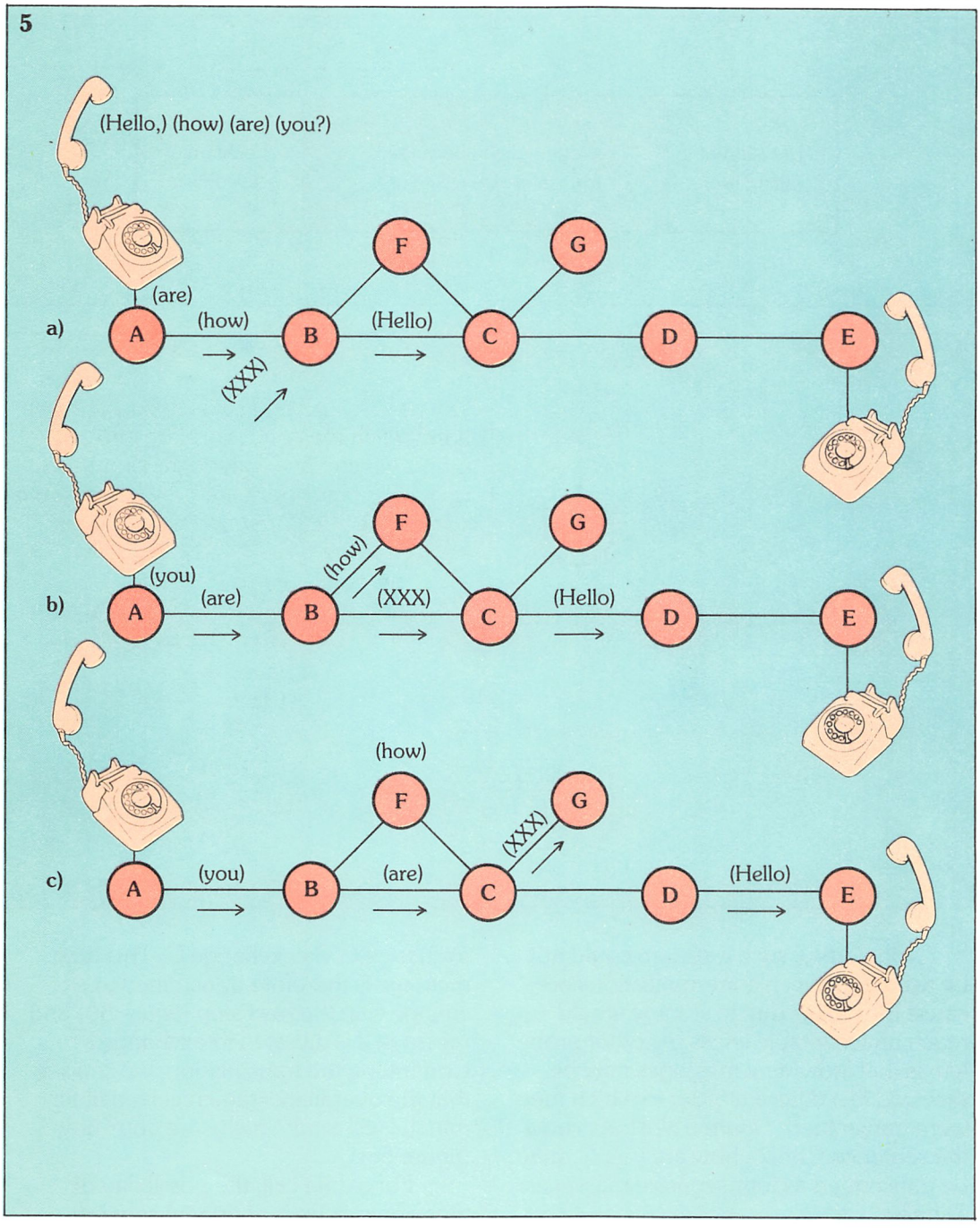
4. Message switched systems provide a memory facility to store messages. They are sometimes referred to as 'store and forward' systems because of this.

Obviously, such a system could not be used for speech communications because the delays due to storage would reach unacceptable levels. Another problem is that individual messages may be received in a different order to which they were transmitted. *Figure 5* illustrates how the sentence: "hello, how are you?", may be transmitted as four separate messages:
1) hello;
2) how;
3) are;
4) you?

Figure 5a illustrates how message 1 has been transmitted from telephone terminal 1 to exchange C, via exchanges A and B; message 2 has similarly been transmitted to exchange B; and message 3 to exchange A. The next step would transmit all messages one exchange further in the sequence. However, before this occurs, a message from another user is received by exchange B, en route to exchange G via exchange C. This new message is therefore transmitted to exchange C in the next step (*figure 5b*) and message 2 is diverted to exchange F. Continuing the transmission, we can see that the overall message received at terminal 2 will read: "hello, are you? how" (*figure 5c*).

For a data call, the possibility of message mix up is of no real concern because the receiving equipment can reorganise the received messages into their correct order, before use. In order to do this however, each message must contain information regarding its position in the stream.

Message switching has advantages over circuit switching because during periods of high system load, the peaks are spread out as messages are stored. As long as sufficient memory is available to store these message peaks, *all* messages transmitted will eventually reach their destina-



5. Transmitting the message "hello, how are you?" as four separate messages.

6. In a message switching system, transmission time increases with system load. However, all messages eventually reach their destinations.

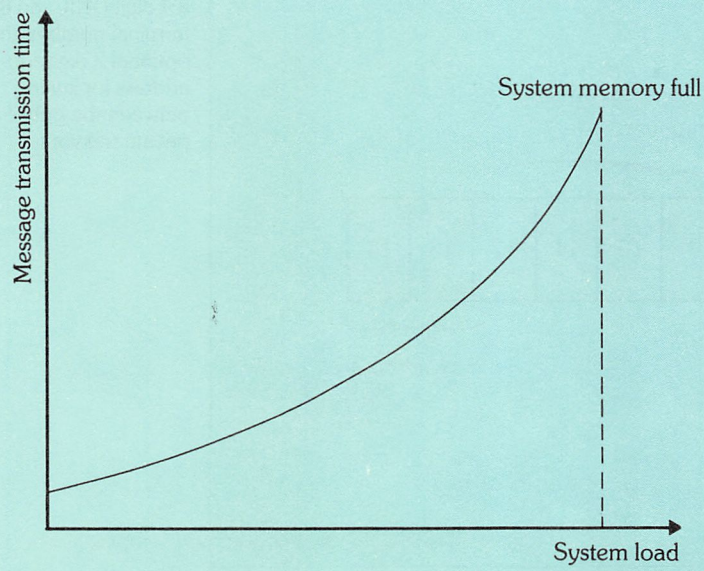
7. Four possible methods of user access to British Telecom's Packet SwitchStream.

tion. In a circuit switched telephone network, on the other hand, the calling party simply hears a system busy tone if the sytsem is fully loaded.

Of course, as the system load increases, more and more messages must be stored and so transmission time increases (*figure 6*). By contrast, increasing system load in a circuit switched system increases call connect time – once the call is connected, however, system load does not affect transmission.

As messages within a message switched system are complete entities, containing all the information necessary to ensure they reach their destinations, they may easily be multiplexed together over the system. It is the system itself which provides the multiplexing facility, unlike circuit switched systems which need add-on devices to perform it. Message switched communications systems therefore have an inherently higher circuit utilisation than circuit switched systems allow.

6



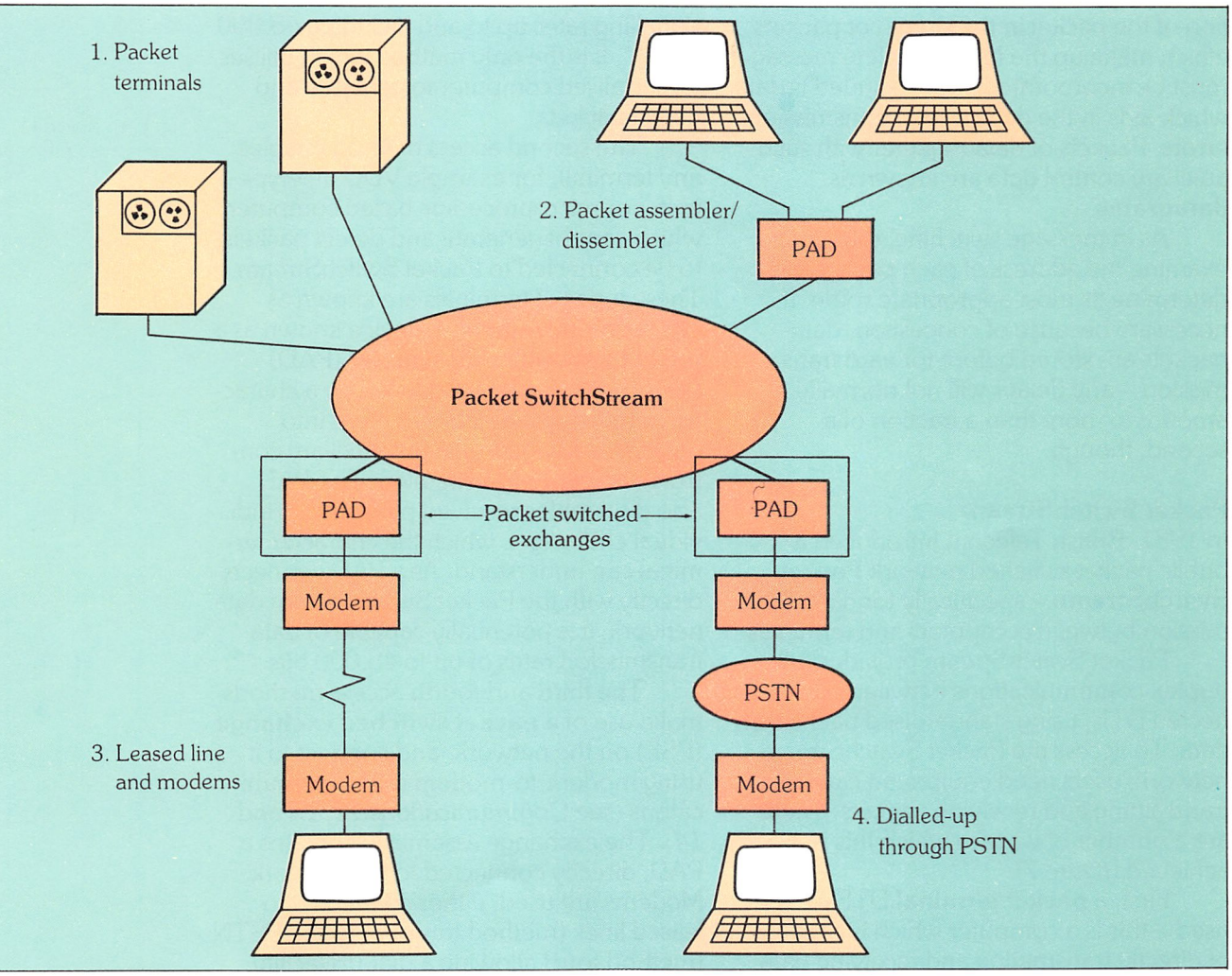
Packet switching

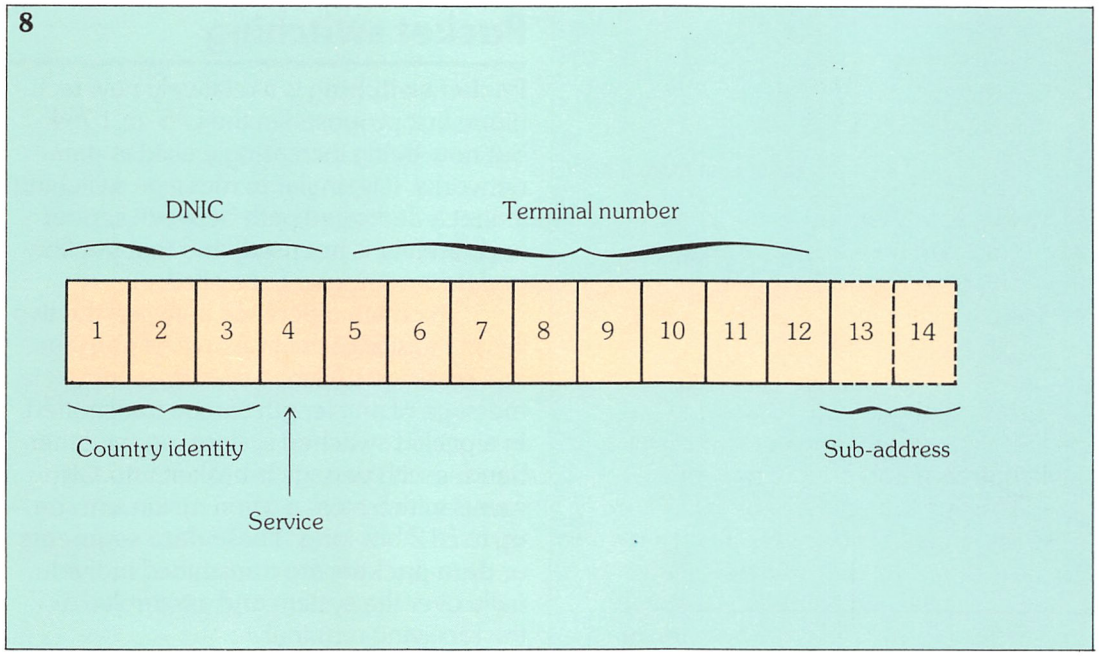
Packet switching is a relatively new technique first proposed in the U.S. in 1964 but now being increasingly used in data networks. It is similar to message switching in that a dedicated path between sender and receiver is not used, and that the flow of data is computer controlled.

The main difference between the two lies in the size of the transmitted message. In a message swtiched system, a complete message of any length can be transmitted. In a packet switched system, on the other hand, each message is broken into segments which have a given maximum size – up to 512 bits long. These **data segments** or **data packets** are transmitted individually over the system and assembled at the receiving terminal.

Additional information must accompany each packet in order that it doesn't

7





8. Network user address (NUA) format comprising a 4 digit DNIC, an 8 digit terminal number and an optional 2 digit sub-address for linking between the public and private networks.

get lost. The identification of the intended recipient and also data regarding the position of the packet in the stream of packets which make up the final complete message must be incorporated. Also included is data which aids in the detection of transmission errors. Packets of data together with such auxiliary control data are known as **datagrams**.

As in message switching, computers examine the address of each packet and determine its most appropriate route. If necessary because of congestion, data packets are stored before forward transmission – any delays will not normally amount to more than a fraction of a second, though.

Packet SwitchStream

In 1981, British Telecom introduced a public packet switched network **Packet SwitchStream** – specifically for data transmission between computers and terminals.

Packet SwitchStream provides full duplex communications between connected DTE, using standardised packets of data. To access the Packet SwitchStream network, users need equipment capable of transmitting and receiving packets. There are a number of ways in which this is achieved (*figure 7*).

First, a **packet terminal** DTE can be used – this is a computer which is capable of directly transmitting and receiving data packets, onto and from Packet SwitchStream. This access method enables data signalling rates up to and including 48,000 bits s^{-1}. It is the only method which utilises a specialised computer to generate and detect packets.

The second access method enables any terminal, for example VDU, teletypewriter or microprocessor-based computer, which cannot generate and detect packets, to be connected to Packet SwitchStream. These types of terminals are known as **character terminals**. A device known as a **packet assembler/dissembler** (PAD) takes the characters produced by a character terminal and assembles them into packets, adding all relevant auxiliary control data before transmission. The PAD also dissembles received packets into individual characters, which the character terminal can understand. As a PAD connects directly with the Packet SwitchStream data network, it is potentially capable of data transmission rates of up to 48,000 bits s^{-1}.

The third and fourth access methods make use of a **packet switched exchange** (PSE) on the network, and connect to it using modem-to-modem data communicatons (see *Communications 12*, *13* and *14*). The exchange essentially contains a PAD, directly connected to the network. Modems are used, either connected to leased lines (method three) or to the PSTN (method four) allowing a dial-up facility,

from remote character terminals. The data signalling rates of both of these methods are limited to the data signalling rates of the modem-to-modem connections.

Addressing terminals

Each terminal attached to the Packet SwitchStream network has a unique address, known as its **network user address** (NUA). All NUAs comprise 12 decimal digits, as shown in *figure 8*. Two additional digits, known as **sub-address digits**, are optionally used to address packet terminals linking between Packet SwitchStream and private networks.

The first four digits of an NUA comprise the **data network identification code** (DNIC – pronounced 'dee-nick'): digits one to three identify the country and digit four identifies the particular service within that country. Digits five to twelve comprise the terminal number within the country and service specified by the DNIC code.

Network user identities

Users access Packet SwitchStream via a dialled-up PSTN connection by supplying a personal code number, known as a **network user identity** (NUI). The NUI prevents unauthorised access.

Speed conversion

Packet switching has one very important advantage over circuit switching systems. Because computers are used to control the switching, and because they also provide for packet storage, the rate at which data is transmitted need not be fixed. This means that the sending terminal can generate packets at a rate which suits its purposes and, similarly, the receiving terminal can accept those packets at its appropriate rate. Devices with very different data signalling rates can therefore communicate with one another.

Packet switching therefore allows for the continual re-allocation of the available bandwidth channel between users, and hence makes the most efficient use of this expensive resource.

International packet switching networks have now been in operation for some years, enabling data flow across national borders. Examples include the European 'Euronet' and the British IPSS (International Packet Switching Service).

Glossary

character terminal	any DTE connected to a packet switched communications system which cannot generate and receive packets
data network identity code (DNIC)	part of user address on the Packet SwitchStream data network which identifies the country and type of service
network user address (NUA)	user address on the Packet SwitchStream data network
network user identity (NUI)	identity code which a Packet SwitchStream user must key in before access is allowed, when dialling-up through the PSTN
packet assembler/ dissembler	equipment which enables character terminals to be connected to a packet switched communications system
packet terminal	any DTE connected to a packet switched communications network which generates and detects packets
store-and-forward	the process by which messages are temporarily stored at the switching centre before forward transmission to the receiver during peak loads or when the terminal is engaged

BASIC THEORY REFRESHER

ELECTRICAL TECHNOLOGY

Digital filters

We know from the previous two *Basic Theory Refreshers* that filters can either be constructed from passive linear networks (which, because of the presence of inductors, will be very large), or from active networks based on operational amplifiers. A third approach to this problem lies in the use of digital filters – these take samples of the input wave and process it to give the output wave.

Analogue filters, remember, are designed by selecting the desired frequency components from the Fourier series representation of the input wave. Later, in the article on convolution, we saw that this procedure was identical to that of determining the convolution of the input time varying function with the impulse response of the network. So a filter, therefore, may be considered as a network which takes the input voltage and processes it in some way to give the output voltage.

Voltage sampling

When using a digital filter, the input voltage is sampled at a number of discrete instants of time, and the values are recorded as numbers. How do you decide on the number of samples to be taken in order to retain *all* the information in the input signal?

This problem was solved by Shannon who found that if a signal contained no frequencies higher than f_m, and if it was sampled at a rate of at least $2f_m$, then all the data would be retained. This corresponds to an interval between samples, T, of $1/2f_m$. We shall assume that all signals here are sampled at this rate.

A representative input voltage is given in *figure 1*: the samples are taken at times 0, T, 2T, 3T, 4T and 5T, and the numerical values of the voltages are given below. A set of binary coded voltages are also given which are defined assuming that the maximum input which can occur is 8 V. Here, 6 bits are used to represent the numbers, so the intervals between each level that can be distinguished is 0.25 V. This is represented by a change of 00001 in the digital representation (since one of the 6 bits is used to represent the sign of the number, leaving 5 bits to indicate the magnitude of the 31 levels above zero).

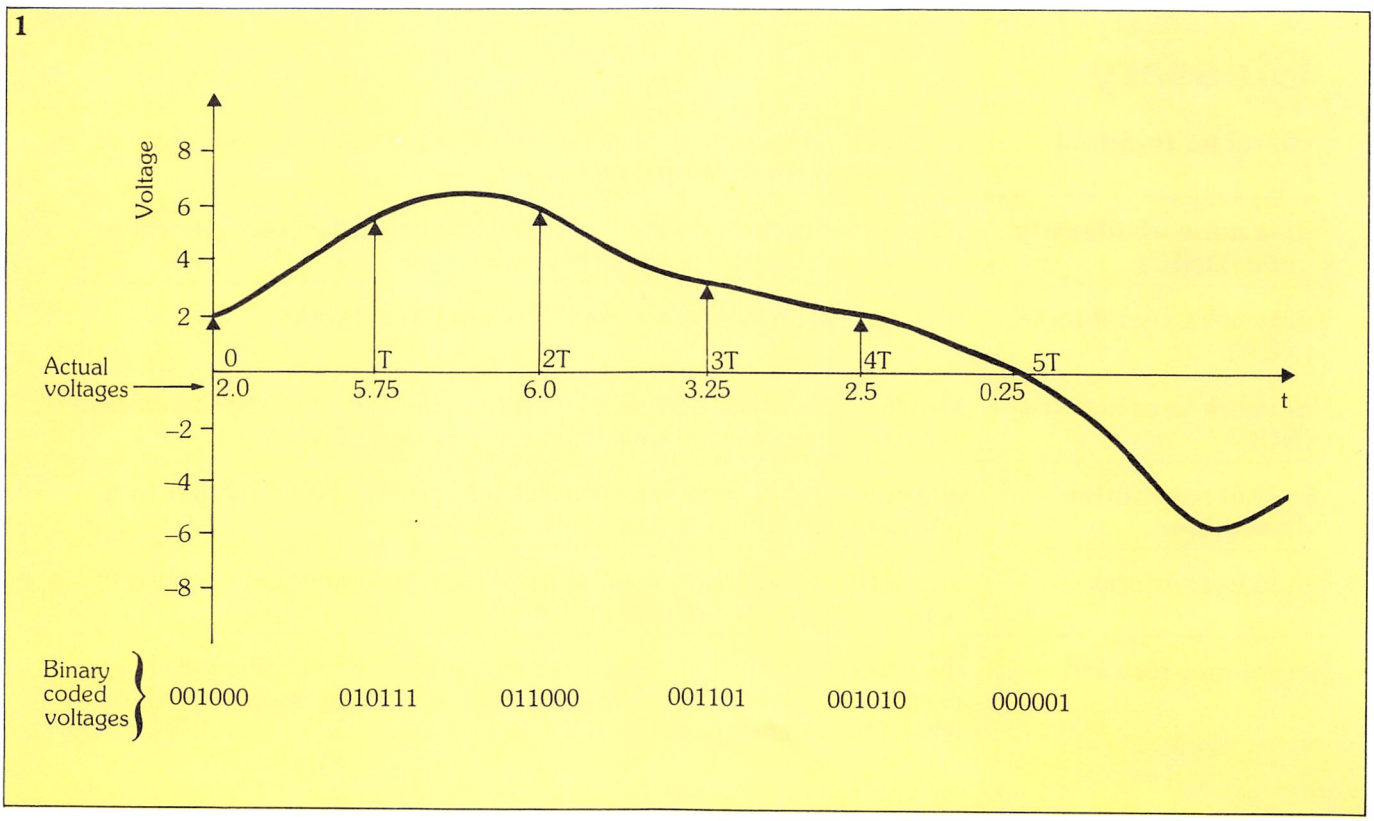
Finite impulse response digital filter

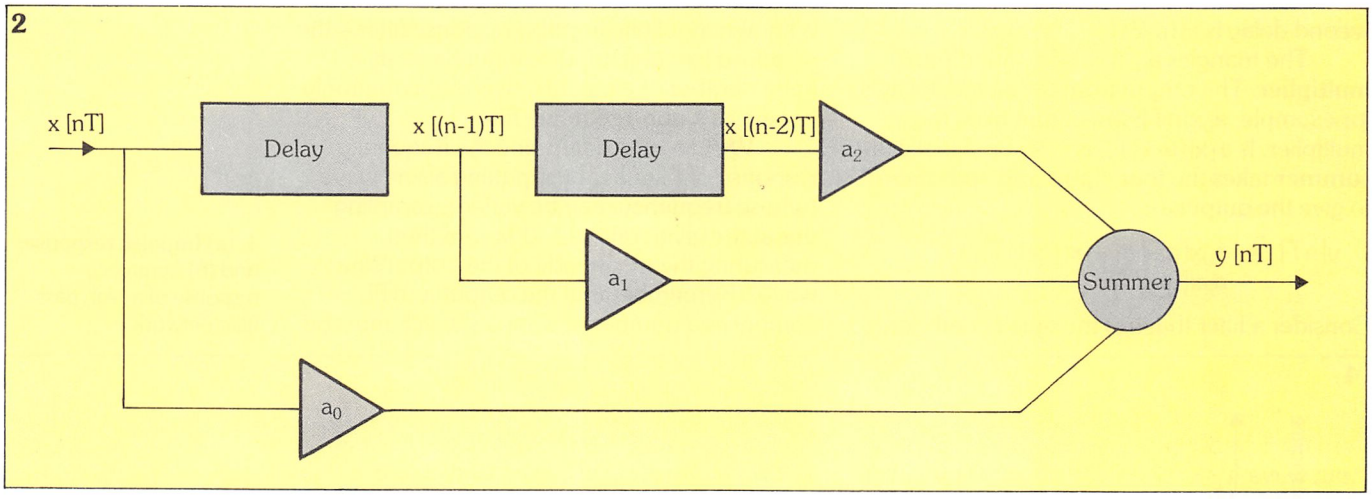
Figure 2 shows an example of a **finite impulse response** (FIR) filter. The input, marked as x[nT], is the sequence of binary numbers which represents the input voltage from *figure*

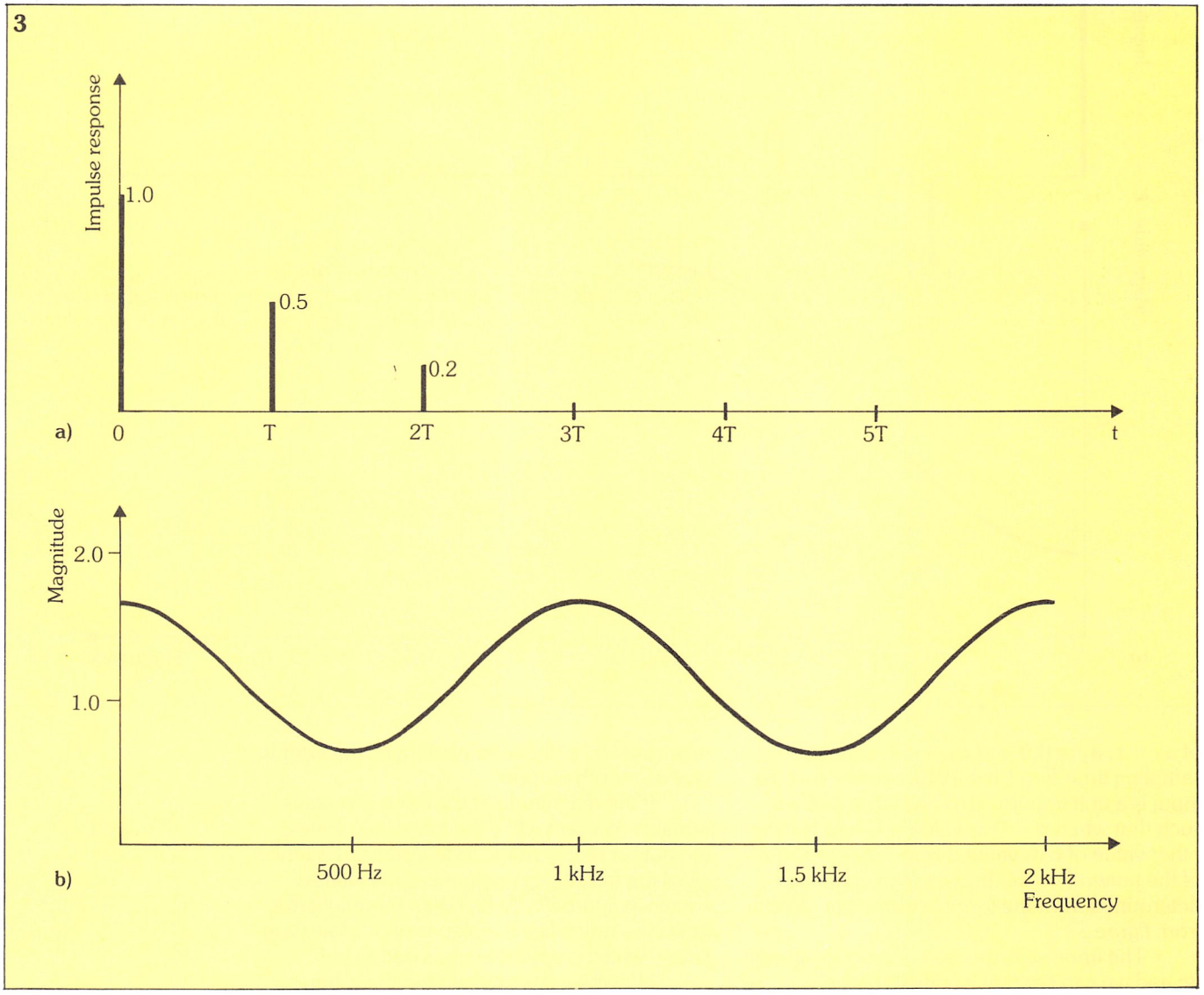
1. A representative input voltage sampled at times 0, T, 2T, 3T, 4T and 5T.

2. A finite impulse response (FIR), or non-recursive, filter fed with the input voltage shown in *figure 1.*

1







3. (a) Impulse response of the network in *figure 2*; **(b)** frequency response of a low pass filter network.

1, taken at the sampling instants of time when n = 0, 1, 2, 3 etc.

The delay boxes take the input number and store it for a delay time equal to one sampling interval. When the input is x[nT], the output of the first delay box will be the value of the input at the previous sampling instant, namely x[(n-1)T]. Similarly, the output of the

second delay is x[(n-2)T].

The triangles a_0, a_1 and a_2 are **digital multiplier**. The output from the a_0 multiplier is, for example, $a_0x[nT]$; the output from the a_1 multiplier, is $a_1x[(n-1)T]$, and so on. Finally the **summer** takes the four inputs and adds them to give the output as:

$$y[nT] = a_0x[nT] + a_1x[(n-1)T] + a_2x[(n-2)T]$$

Consider a filter that has multiplier coefficients of $a_0 = 1$, $a_1 = 0.5$ and $a_2 = 0.2$, and a sampling time T = 1 ms. We'll assume that the input is a unit impulse x[nT] which is defined such that when n = 0 its value is 1, and for any other value of n its value is zero. The response of the network to this impulse can either be determined from the formula above, or directly from *figure 2*.

The impulse response is shown in *figure 3a* and we can see that it consists of three elements: the first of value 1 at time 0; the second of value 0.5 at time T; and the third of value 0.2 at time 2T. After this time, the output is zero. It is for this reason that this type of filter is known as a finite impulse response filter – the response to a unit impulse input has only a finite number of elements. (It is also commonly known as a **non-recursive filter**.)

We can now determine the frequency response of this filter by inputting a sine wave (whose frequency may be varied), sampling this at the given rate of 1 kHz, and then measuring the magnitude of the output sine wave. (Remember that the output, y[nT], comprises a number of samples which must be processed by a digital-to-analogue converter to give a smooth output.)

If the magnitude of the input sine wave remains constant while the frequency varies, the output plot of magnitude against frequency gives the frequency response of the filter as shown in *figure 3b*. At first sight this might not look very much like the response of a low pass filter – which is what it is supposed to be!

However, if we remember that Shannon's theroem limits the input waveform to one which does not contain frequencies higher than half the sampling rate (for reasons of satisfactory reproduction), then we would know that

4. (a) Impulse response; and (b) frequency response of a high pass filter network.

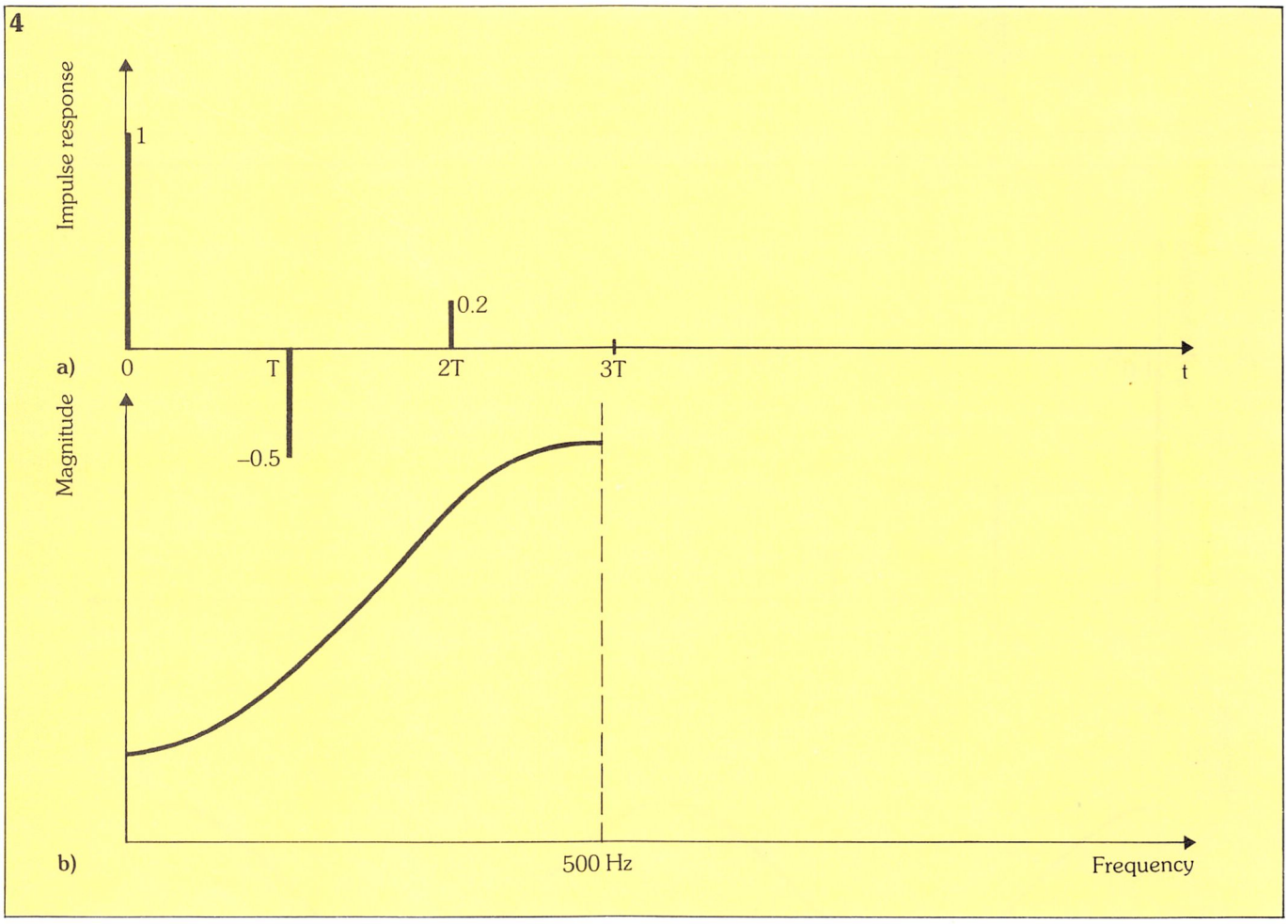


Table 1
Output values according to some arbitrary input voltage waveform

Input	Output at time						
	0	T	2T	3T	4T	5T	6T
2.0	2.0	–1.0	0.4				
5.75		5.75	–2.875	1.15			
6.0			6.0	–3.0	1.2		
3.25				3.25	–1.675	0.65	
0.25					0.25	–0.125	0.05
etc.							
Total output	2.0	4.75	3.525	1.40	–0.225		

this filter cannot be used for frequencies higher than 500 Hz. Within this range, then, it acts as a very satisfactory low pass filter.

We can now see that for satisfactory operation the signal must be band limited to half the sampling rate of the digital filter. Filters like this are extremely simple to construct, and the concept may be extended to include many more delay sections – up to 50 not being uncommon.

High pass filters

We'll now look at a filter almost identical to the one above except that the sign of the second coefficient is reversed, giving $a_0 = 1$, $a_1 = -0.5$ and $a_2 = 0.2$. The impulse response of this filter is shown in *figure 4a* and its frequency response, over the range 0 to 500 Hz, is given in *figure 4b*; it therefore represents a high pass filter over this range.

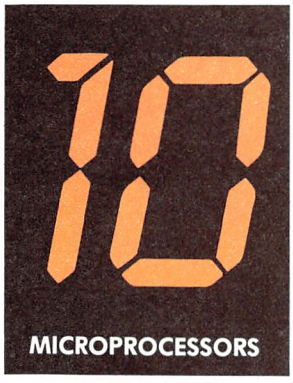
This shows us one of the great advantages of digital filters: by altering just one coefficient (which is merely a number, possibly stored in ROM), the filter's performance can be dramatically altered. In more complex circuits, all the coefficients may be changed producing filters of various types with variable pass bands and cut-off frequencies. This provides an almost infinitely variable filter, which can be controlled by a computer.

Determining the filter's output

We can now go on to see how the output from a filter like this can be determined, when the input is some arbitrary waveform, like the one in *figure 1*. This is represented by the sequence of numbers 2.0, 5.75, 6.0, 3.25, 2.5, 0.25... We'll assume that this wave is filtered by the high pass filter whose impulse response is given in *figure 4a*, and is represented by the sequence 1, –0.5, 0.2, 0, 0...

The output to the input value 2.0 occuring at t = 0 as the sequence 2.0, –1.0, 0.4, may be determined by multiplying the input response by the input value at time t = 0. This response starts at t = 0 and goes on to t = 2T (after which all the values are zero and have been omitted). Similarly, the response to the input 5.75 occuring at t = T is the sequence 5.75, –2.875, 1.15, starting at t = T and finishing at t = 3T. These values can be shown as in *table 1*.

The output magnitude at any instant of time is obtained by adding together the responses due to each element of the input. This gives the sequence shown in the bottom line of the table: 2.0, 4.75, 3.525, 1.4, –0.225. If these values are represented as voltages, and smoothed to give a continuous wave by a digital-to-analogue converter, we have a wave which is the high pass filtered version of the wave in *figure 1*. □



Microprocessor programs

Programming languages

No matter which microprocessor is chosen for a given task, the programs must be stored in the system's memory. Programs are generally written in a problem oriented language to simplify development, but are stored as machine code.

Because machine code is not readily understandable, programs are usually written in assembly language (mnemonic form) or an even higher level language. You'll remember that converting a high level language to assembler is known as **compiling**; while converting assembly language program to machine code is **assembling**. These are both tasks that can be performed by software routines or, in the case of assembling, sometimes even manually.

Assembly language programming

We have already seen several examples of assembly language and the mnemonic forms from which it is composed. For a better understanding of the effort involved in assembly language programming, and how that effort is dependent on the instruction set used, we'll consider another example.

Two 32-bit binary numbers are to be added, and the result stored in place of the second number. It is also assumed that the hardware design has fixed the location of data memory at 8000_{16} to $8FFF_{16}$, and that it has been decided to locate the first 32-bit number in locations 8000_{16} to 8003_{16}, and the second 32-bit number (and the sum) in locations 8010_{16} to 8013_{16}. Each location stores an 8-bit byte, so four locations are needed for a 32-bit number.

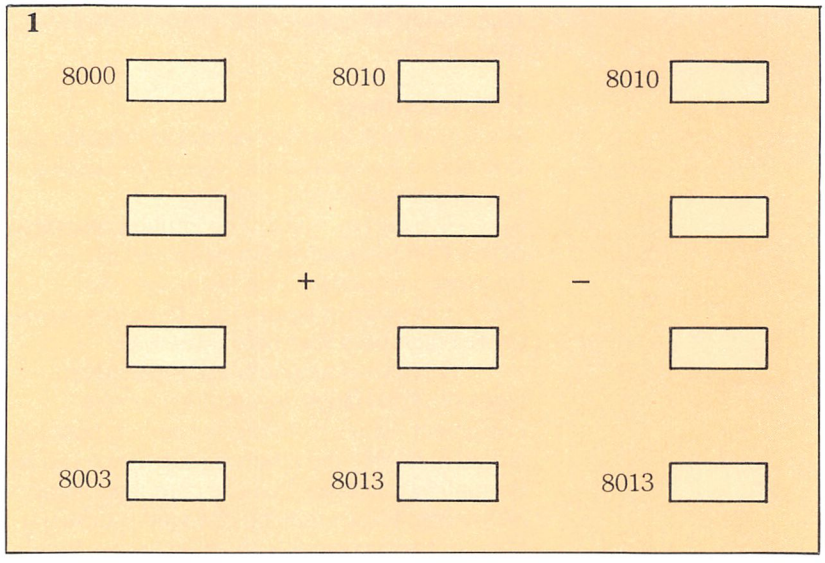
The algorithm used to perform this addition is the same procedure that we would use to add the numbers on paper. The least significant group of bits are added first, and any resulting carry is noted. The next least significant bits are added with any previously generated carry as an input. This process is repeated until the most significant group of bits and their input carry have been added.

This procedure is represented in terms of memory locations in *figure 1* and a flowchart in *figure 2a*, which implies a repetitive loop structure. This would best be implemented with a program loop, in the case of the 4-bit and 8-bit microprocessors. A program loop would not be necessary for the 16-bit device though.

Figure 2b shows the TMS8080A program that performs this task, almost directly related to the flow chart on a one to one basis.

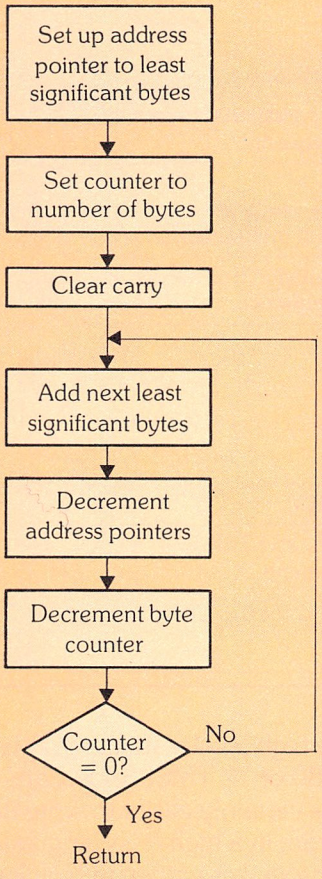
The H and D registers are initially set to the address of the least significant bytes of the two numbers, and a loop counter (C register) is initialised to 4 (for the four bytes to be added). The carry is cleared with the XRA A instruction, after which the repetitive loop is entered. If the Exclusive OR function, bit by bit, is performed by the accumulator on itself, the accumulator is cleared to zero.

1. 32-bit addition memory structure.



2. (a) Flowchart description; and **(b)** TMS8080A program for 32-bit addition.

2



a)

LABEL	INST	OPERAND	BYTES
	LXI	H,8013H	3
	LXI	D,8003H	3
	MVI	C,4	2
	XRA	A	1
LOOP	LDAX	D	1
	ADC	M	1
	MOV	M,A	1
	DCX	H	1
	DCX	D	1
	DCR	D	1
	JNZ	LOOP	3
	RET		1
			19

H indicates hexadecimal
12 assembly language statements
19 bytes of program memory

b)

3. TMS9980 16-bit microprocessor program for 32-bit addition.

3

```
        LWPI    >8000
        A       R1,R9
        JNC     S1          High Level Language Equivalent
        INC     R8              N = N + M
SI      A       RO,R8
        RT
```

> Indicates hexadecimal
6 Assembly Language Statements
16 Bytes of Program Memory

Inside the loop, the first byte of the first number is moved into the accumulator with the LDAX D, and the first byte of the second number is added to it with the ADC M. The byte sum is sent back to the second number location with the MOV M, A operation. After this, the address registers are decremented to indicate the next least significant bytes and the loop counter is decremented.

If the loop counter has been decremented to zero, the addition is complete; otherwise, the loop is repeated. This program requires 12 assembly language statements. Each of these instructions requires either 1, 2 or 3 bytes of program memory as shown in *figure 2b*. The machine code version of this program therefore requires 19 bytes in program memory.

For comparison, the TMS9980 version of a 32-bit binary addition is shown in *figure 3*. The TMS9980 acts like a 16-bit microprocessor, even though it actually operates on data one byte at a time. It processes two bytes per instruction.

In this case, the address is set initially by loading a register called the **workspace pointer** with 8000_{16}, by using the LWPI>8000 instruction. No loop counter is needed so locations 8002_{16} and 8003_{16} can be added to locations 8012_{16} and 8013_{16} with the A R1, R9 instruction. The JNC S1 checks for a carry to the next 16-bit addition.

4

a)

Language	Multiplication	Decisions
FORTRAN	Z = W*Y	IF (condition) GO TO condition true sequence condition not true sequence
PASCAL	Z = W*Y	IF condition THEN condition true sequence ELSE condition not true sequence
BASIC	Z = W*Y	IF condition THEN condition true sequence condition not true sequence

b)

FORTRAN	PASCAL	BASIC
DO 10 I = 1,N Loop statements 10 CONTINUE	I = 1 WHILE I < = N D0 BEGIN Loop Operations END	FOR N = 1 TO 25 STEP 1 Loop Statements NEXT

4. Comparison of formats for the three high level languages, PASCAL, FORTRAN and BASIC: **(a)** arithmetic and decision making; **(b)** loop structures.

If there is a carry, 1 is added to the second 16-bit number (most significant 16 bits of the 32-bit number) located in workspace pointer register R8 with the INC R8 instruction. Then the most significant 16-bit groups are added with the A RO, R8 instruction.

This program requires only 6 assembly statements – a two-to-one reduction over the requirements for the 8-bit microprocessor program. The 16-bit unit only requires 16 bytes of program memory to store the machine code for the program in *figure 3*. The advantage of processing data 16 bits per instruction is evident even in this simple example – 6 assembly language statements and 3 bytes of memory are saved.

High level languages

Although use of a longer bit microprocessor may simplify the assembly language programs, using a high level language to write the programs can simplify the task even further.

FORTRAN, BASIC and PASCAL are all high level languages, and typically in each language the add operation performed by the assembly language program in *figure 2b* can be simply written as:

N = N + M

So, a single high level statement replaces six TMS9980 or 12 TMS8080A commands – it's certainly easier to write the initial program in a high level language. However, programs written in high level languages have to be translated into assembly language by means of a computer program, and then changed into machine code with an assembler.

Figure 4 shows some further examples of typical high level language statements. *Figure 4a* details their arithmetic and decision making formats, while *figure 4b* shows the different loop structures. All of the three example languages use the same single line statement for multiplication. Their decision statement structures are similar and allow reasonably complicated arithmetic and logical tests to be made to decide which of the two program sequences are to be executed.

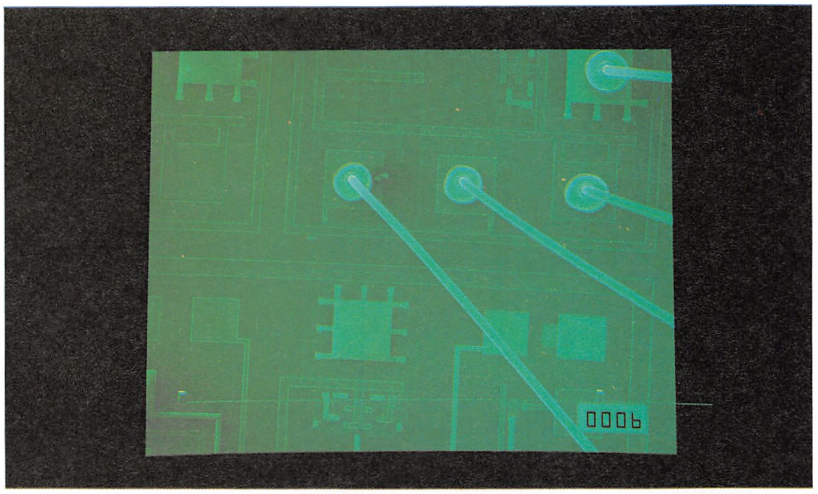
The loop structures too, are very similar.

All of these operations would have asembly language equivalents for any given microprocessor. Again, the advantage of the high level language is that a system subprogram can be described in a short, easy-to-understand high level language program, instead of a lengthy assembly language program.

Table 1
Program verification options

	Program entry form	Execution
A	Hexadecimal machine code entered into microcomputer board	Machine code program executed on computer
B	Assembly language code entered into computer	Assembly and machine code program executed on computer
C	High level language code entered into computer	Compiling, assembly, and execution on computer
D	Assembly language code entered into general purpose computer	Assembly by computer; simulation of microprocessor execution by general purpose computer or execution of machine code on actual microprocessor
E	High level language code entered into general purpose computer	Compiling and assembly done on general purpose computer. Execution simulated on general purpose computer or executed on actual microprocessor

Below: scanning electron micrograph of an IC chip showing wire bonding.



Neville Miles

Program verification

Regardless of the level of language used to write the system program and subprograms, the programmer must verify their operation before they can be used to control the computer system.

Table 1 lists the options available for program verification. The most tedious of these options (A) is where the entry of the machine code is made directly into memory and execution is then performed on the computer. The programmer must manually assemble the program and then enter the hexadecimal machine codes for the program into the computer.

Option B is a more convenient approach. This can be carried out using available microprocessor based **program development boards** or any of the **computer development systems**. The assembly language programs are entered into the computer in mnemonic form and the computer performs program assembly and execution.

Some development systems also offer the option (C) of entering the desired program in high level language form. The computer then performs the compiling, assembly, and execution of the program. This offers the most efficient way to develop programs as far as time taken is concerned.

If (as shown in D and E) a microprocessor based computer is not available, many manufacturers provide program development software to run on general purpose computers that will perform the assembly and compiling operations. Once the machine code has been generated, the general purpose computer then executes the machine code program in just the same way as the microprocessor based computer.

This is known as **simulating** the execution of the program and it also verifies the program operation. The only problem that may arise from this type of program verification is that the program does not run in the actual environment of the final computer system. So some timing or interrupt problems may not be detected by this method.

Typical program requirements

Certain types of subprogams have to be developed for all systems, for example some form of input and output subprograms. Usually the system has to support some arithmetic and logic operations and a decision or 'look-up' table may be required.

We'll now look at some typical examples of these operations, to illustrate the relationship between high level language programs and the corresponding microprocessor programs.

Input/output subprograms

When using SAM, we know that BCD codes must be sent as outputs to 10 LED displays to display 10 decimal digits. We can now use this task as an example output subprogram.

The BCD information is located in 10 successive memory locations, and the LEDs occupy 10 successive addresses. The programs for accomplishing this transfer are shown in *table 2*. There is no difference between the lengths of the TMS1000 and the TMS8080A assembly language programs: each requires 10 statements. The TMS1000 program occupies 10 program memory locations while the TMS8080A program requires 17 bytes of program memory. The TMS9900 program can be written in only 6 assembly language statements since no program loop is involved. It requires 16 bytes of program memory.

In contrast, the equivalent BASIC statement is a single line statement, which certainly is a more efficient way to summarise the program being written.

Table 2
Programs for transferring BCD information to LED displays

TMS1000 Program		TMS8080A Program			TMS9900 Program		
OUTPUT	LDX 2	OUTPUT:	LXI	D,200H	OUTPUT:	LI	R2,>200
	TCY 0		LXI	H,300H		LI	R3,>300
LOOP	TMA		MVI	C,5		MOV	*R2+,*R3+
	TDO	LOOP:	LDAX	D		MOV	*R2+,*R3+
	SETR		MOV	M,A		MOV	*R2+,*R3 +
	RSTR		INX	D		RT	
	IYC		INX	H			
	YNEC 10		DCR	C			
	BR LOOP		JNZ	LOOP			
	RETN		RET				
Assembly language statements:	10		10			6	
Machine code memory locations:	10		17			16	

Arithmetic operation

An example of an arithmetic operation is multiplication. If the system requires the multiplication of one 16-bit binary number by a second 16-bit binary number to yield a 32-bit binary product, the programs would look like those in *table 3*. The TMS1000 version is not shown, since it is much too long to be clearly understood. Again, the high level language statement takes only one line. In this case, the TMS9900 assembly language subprogram only takes one line, since this is a single instruction of its instruction set. By comparison, the TMS8080A multiplication program requires 37 assembly language

Table 3
Comparison of applications programs

TMS8080A Program			TMS9900 Program	High Level Language
	MVI	B,16	MPY R2,R4	Z = Z*Y
	CALL	CLEAR		
LOOP:	CALL	SRT		
	JNC	ARND		
	CALL	ADDER		
ARND:	DCR	B		
	JNZ	LOOP		
	CALL	SRT		
	RET			
CLEAR:	LXI	H,PROD		
	MVI	C,2		
	XRA	A		
LPC:	MOV	M,A		
	INX	H		
	DCR	C		
	JNZ	LPC		
	RET			
SRT:	LXI	H,PROD		
	MVI	C,4		
LPS:	MOV	A,M		
	RAR			
	MOV	M,A		
	INX	H		
	DCR	C		
	JNZ	LPS		
	RET			
ADDER:	LXI	D,MCNDL		
	LXI	H,PRDL		
	LDAX	D		
	ADD	M		
	MOV	M,A		
	DCX	H		
	DCX	D		
	LDAX	D		
	ADC	M		
	MOV	M,A		
	RET			

statements which would require 64 bytes of program memory.

The flowchart for the TMS8080A multiplication program is shown in *figure 5*. The algorithm is much the same procedure that one would follow if the multiplication were being performed by hand on a sheet of paper. Even without going through this program in detail, it is easy to see the advantage of the more powerful instruction set offered by the 16-bit microprocessor.

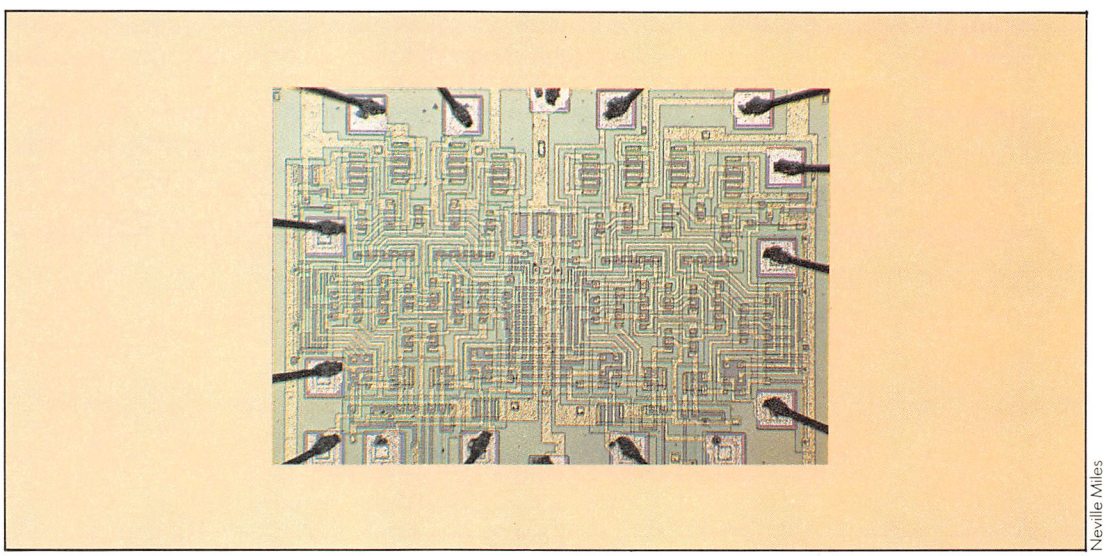
Conclusion

We have seen that program design for microprocessors is very much like hardware design: design effort is broken down into manageable functional parts and the design for each part is completed before the whole program is combined.

The higher the level of programming language used to write a program, the simpler it is to write and verify. However, a much greater amount of software aid is required as support for the product.

Assembly language programs are shorter and more efficient for microprocessors of longer bit-length, because they have a bigger and better instruction set. Program requirements at any given level may be described in terms of the general system model by defining the module inputs, outputs and operations.

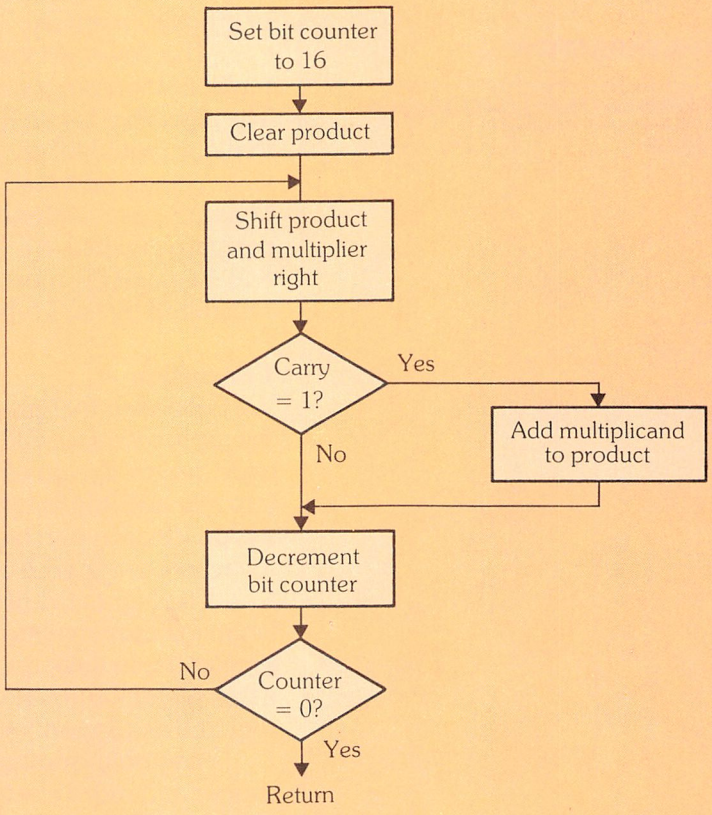
Finally, program requirements may be described in a variety of forms, with the flowchart and macro-mnemonic coding forms being the most directly related to the final program organisation.



Right: photomicrograph of a 7473 flip-flop showing wire bonding.

Neville Miles

5



5. Flowchart for the TMS8080A program multiplying 2 16-bit numbers to give a 32-bit binary product.

Glossary

computer development system	a system (generally microprocessor based) which is used to enable other computer systems to be developed. Hardware and software can be designed and developed and it allows the final system to be simulated without being fully built
program development boards	printed circuit boards containing a microprocessor, which may be used to develop a program to be run on a microprocessor based system
simulate	use of general purpose computer program development boards, or computer development system, to verify program operation, while not actually running the program on the system for which it is intended
workspace pointer	a register assigned the task of pointing to a sequence of instructions held in memory